SELF-ORGANIZING DISTRIBUTED WORKFLOW MANAGEMENT

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Nenad Stojnić
aus Zagreb, Croatia

Basel, 2015

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät

auf Antrag von

Prof. Dr. Heiko Schuldt, Universität Basel, Dissertationsleiter
Prof. Dr. Cesare Pautasso, Università della Svizzera italiana, Korreferent

Basel, den 21.04.2015

Prof. Dr. Jörg Schibler, Dekan

Zusammenfassung

Die Verbreitung von Service-orientierten Architekturen in den letzten Jahren hat eine wichtige Klasse von anspruchsvollen und verteilten Anwendungen hervorgebracht welche auf der Idee beruhen, mehrere und einfache Dienste in ein komplexes, zusammenhängendes Ganzes zu kombinieren. Solche Anwendungen, die sich über mehrere Service-Aufrufe erstrecken, können am effektivsten mit der Hilfe von Workflows umgesetzt werden. Wenn es um die Hochleistungsausführung von Workflows geht, ist die Verteilung (Hochskalierung) von Diensten ein Schlüsselkonzept und auch eine gegebener Vorteil des Workflow-Paradigmas. Konkret bedeutet das, dass sowohl die beinhalteten Dienste des Workflows als auch die Dienste des Systems, welches ihre Aufrufe verwaltet, auf eine Menge von Rechenknoten verteilt werden müssen. In einem breiten Spektrum von Anwendungen die Heterogenität der umfassten Rechenknoten nach sich ziehen, z.B. im modernen Notfallmanagement, sind Aufrufe von optimalen Dienstinstanzen sowie deren Zuverlässigkeit Grundvoraussetzungen des verteilten Workflow-Managements.

Das Hauptaugenmerk der Arbeit ist ein formales Modell, welches die verteilte (d.h. skalierbare) Ausführung von Workflows definiert. Um dieses Modell auf neuartige Weise um Zuverlässigkeit zu erweitern, ohne die Skalierbarkeit der Ausführung negativ zu beeinflussen, wird der Systemdienst “Safety-Ring” vorgestellt. Die Idee hinter Safety-Ring ist die Wiederherstellung einer Vielzahl von Knotenausfällen, welche aktive Dienste von laufenden Workflows beherbergen. Zu diesem Zweck bietet Safety-Ring einen skalierbaren, zuverlässigen und konsistenten Datenspeicher, der für die Speicherung von Workflow-Ausführungszuständen verwendet wird. Der neuartige Wiederherstellungsmechanismus von Knotenausfällen bietet hohe Zuverlässigkeit, so dass er auch für die Knoten, welche den Safety-Ring-Dienst selbst anbieten, angewendet werden kann. Deshalb nennen wir den Safety-Ring “selbstheilend”.

Um das zuverlässige (und verteilte) Ausführungsmodell (durch Safety-Ring erweitert) auf heterogene Knotenumgebungen, die überwiegend aus mobilen Geräten zusammengesetzt sind, anzuwenden, führt diese Arbeit das Compass Datenzugriffsprotokoll ein. Bei der Bereitstellung der skalierbaren Datensuche für seine verwalteten Daten setzt Safety-Ring stabile Laufzeitcharakteristiken des Netzes voraus. Somit optimiert Safety-Ring implizit die Anzahl der abgefragten Knoten. Insbesondere bei mobilen Anwendungen, bei denen sich die Netzwerkverbindungen der Knoten dynamisch verändern, sollte das Datenzugriffsprotokoll bei der Datensuche auf eine Verringerung der Latenz anstatt der Anzahl der abgefragten Knoten zielen. Kompass führt latenzoptimale Pfade zu jedem Knoten ein, die sich dynamisch an die verändernden Netzwerkeigenschaften in der Umgebung anpassen. Die Skalierbarkeit der Datensuche von Safety-Ring wird dabei nicht negativ beeinträchtigt.

Im Falle der Ausweitung des verteilten Ausführungsmodells auf Diensttypen, die kontinuierlich und zustandsbehaftet sind, wird Zuverlässigkeit von Safety-Ring nicht gewährleistet. Da solche Diensttypen überwiegend von Geräten mit begrenzten Ressourcen angeboten werden, sollten neue Ansätze zur ressourcenschonenden Wieder-

herstellung von Fehlern der kontinuierlichen Dienste vorgesehen werden. Diese Arbeit basiert auf bewährten Wiederherstellungstechniken wie dem passiven Standby, um sie für die Redundanz der kontinuierlichen Zustände zu erweitern und somit die Gesamtzuverlässigkeit des Systems zu verbessern. Dabei wird die Redundanz der Zustände durch ein leichtgewichtiges (bezüglich des Netzwerk-Overheads) Konsistenz-Protokoll durchgesetzt so dass die Anwendung in Umgebungen mit ressourcenbegrenzten Knoten ermöglicht ist.

Um den Durchsatz der verteilten Workflows zu verbessern, stellt diese Arbeit ein neuartiges Konzept zur Verteilung der Dienste vor. Das Herzstück unseres Ansatzes beruht auf dezentralen Controllern, die selbstständig die dynamische Rekonfiguration der Ausführungsumgebung im Hinblick auf die verfügbaren Dienste vornehmen. Dies betrifft vor allem den Safety-Ring Dienst sowie alle Anwendungsdienste. Dabei sind die Ziele der Controller die Vermeidung von Engpässen bei der Workflow-Ausführung sowie unnötige Dienstverteilungen die Ressourcen verschwenden. Da die Controller an jedem beliebigen Knoten im System aufgesetzt sind und jeden anderen Knoten im System beeinflussen können sagen wir, dass das verteilte Ausführungsmodell “selbstoptimierend” ist.

Schlussendlich bietet diese Arbeit eine Implementierung aller vorgestellten Konzepte im Rahmen der verteilten Workflow-Engine OSIRIS sowie ein quantitative Evaluierungen mittels einer Reihe von Experimenten. Die Ergebnisse der Experimente bestätigen den Mehrwert der Konzepte für das verteilte Ausführungsmodell der Workflows.

Abstract

The proliferation of service-oriented architectures in the last decade has brought forward an important class of sophisticated distributed applications that are founded on the idea of composing multiple simple services into a complex, coherent whole. Such applications spanning multiple service invocations can be most effectively realized by means of workflows. When it comes to high performance workflow execution, distribution (outscaling) of services is a key concept and also a very straightforward advantage of the workflow paradigm. Concretely, both the constituent services of the workflow and the system that manages their invocations have to be distributed across an environment of computational devices. In a wide spectrum of applications, that entail heterogeneity of the encompassed computational devices, e.g., modern emergency management, invocations of optimal service instances in conjunction to their reliability are fundamental prerequisites of distributed workflow management.

At the center of this thesis is a formal model that defines the distributed (i.e., scalable) execution of workflows. To extend this model for reliability in a novel way, which does not affect the scalability of execution, the Safety-Ring system service is presented. The idea behind Safety-Ring is to offer recovery for a wide range of node failures, which host active services of running workflows. To this end, the Safety-Ring provides a scalable, reliable, and consistent data store that is used for the storage of workflow execution state. The novel failure-recovery mechanism features effective reliability such that can be applied on the nodes that host the Safety-Ring service themselves, thus we say the Safety-Ring is self-healing.

To apply the reliable (and distributed) execution model, enhanced by Safety-Ring, to heterogeneous node environments, that are predominantly composed of mobile devices, this thesis presents the Compass data access protocol. In providing scalable data lookup for its maintained data, the Safety-Ring assumes network runtime characteristics which are rather stable, and thus Safety-Ring implicitly optimizes for the number of queried nodes. Especially in mobile applications, where node network connectivity dynamically changes, data access protocols should aim at reducing the overall data lookup latency, rather than the number of queried nodes. Compass introduces latency optimal paths to each node, which dynamically adapt to changing network characteristics. The scalable data lookup of Safety-Ring is not affected.

In case distributed execution of workflows spans services of continuous (stateful) type, their reliability is decoupled from the Safety-Ring. Since such service types are predominantly featured by devices of limited resources, novel approaches to resource conservative recovery of failures for continuous services have to be provided. This thesis builds on proven recovery techniques, such as passive-standby, so as to enhance them for redundancy of the continuous state and thus improve the overall reliability of the system. In doing so the redundancy of state is enforced by means of a lightweight, in terms of network overhead, consistency protocol which allows for its application in resource limited node environments.

In order to improve the execution performance of distributed workflows, in terms of throughput, this thesis offers a novel concept to services distribution. At the heart of our approach lie decentralized controllers that autonomously perform dynamic reconfiguration of the execution environment in terms of available services. This primarily affects the Safety-Ring service and all application services. Thereby, the goals of the controllers are to prevent workflow execution bottlenecks and unnecessary service deployments that waste resources. Since the controllers are equipped at any node in the system and can affect any other node of the system we say that the distributed workflow execution model is self-optimizing.

Finally, all the presented concepts are implemented within the context of the OSIRIS distributed workflow engine and quantitatively evaluated in a series of experiments. The results of experiments confirm the benefits of our concepts for the distributed workflow execution model.

Acknowledgements

Firstly, I want to thank my advisor, Prof. Dr. Heiko Schuldt, for giving me the opportunity to obtain a Ph.D. in the DBIS group. His guidance and supervision throughout my whole time at DBIS group, have been very helpful, friendly and have always inspired me to improve my work far beyond levels previously thought to be impossible.

Secondly, I wish to thank my co-reviewer, Dr. Cesare Pautasso from the University of Lugano, for his willingness to review my thesis and for his time and effort in doing this.

I wish to thank my former and present colleagues of the DBIS group for the many interesting discussions and mutual support over the last years. Out of the DBIS members I would especially like to thank Filip, Ilir and Ihab for the valuable collaboration from the very beginning. Moreover, I would like to thank Diego for helping me to start my work which resulted in this thesis. It has been a privilege to spend time and learn from so many knowledgeable people.

A special thanks goes to the colleagues of the other groups, i.e., Ghazi, David, Dinu, Manos and Behrouz, for the many interesting hours outside of work, especially at the kicker table. I am very grateful to all of my other friends, i.e., Aleks, Dražen, Peđa, Gvozden and Đole for convincing me to do a Ph.D and for supporting me from early on.

My biggest gratitude however goes to my parents Brankica, Slobodan and my brother Mladen for giving me everything in life. Without them this thesis would have not been possible. This thesis is dedicated to them.

Finally, I want to thank my beloved Vanja for being with me and supporting me endlessly with love in the final and most difficult stages of my student days.

I thank you all.

Contents

List of Figures

List of Tables

1

Introduction

The remarkable price reductions of commodity hardware in the last couple of years have led to an unprecedented proliferation of computational devices in our every day lives. Their gradual improvement in computational power along ever increasing adoption of open software standards have made it easily possible to combine such devices into notable clusters of powerful computational resources at a low cost. As it turns out such loosely coupled computational environments are perfectly suited for the innovation of applications of rather complex functionality. By merging smaller, simpler and already existing applications offered at a variety of available commodity devices into a big coherent whole sophisticated applications can be created. The traditional approaches to handling complex functionality are mainly based on centralized, single platform and monolithic architecture programs. In general monolithic architecture solutions incur powerful and dedicated supercomputers that are costly to procure and maintain. Hence, modern innovations and newly added values are based on the idea of available application integration and have the upper hand over traditional monolithic architectures.

From a software architecture integration point of view, complex high-level applications can be achieved by means of Service-oriented architectures (SOA). The Service-oriented computing [ACKM10, SH05] paradigm abstracts an atomic set of computational instructions into a self-contained activity or formally application *service*. The self-contained activity (i.e., service) is enclosed with open-standard and cross-platform interfaces that allow for an easy communication over a computer network and thus integration with other services. In turn, computational devices of any hardware characteristic (in terms of CPU and memory power) and physical location capable of conducting the underlying computational instructions can offer activities for serving at their convenience to high-level applications. They merely have to register themselves and their hosted services conditions to publicly accessible service repositories. Moreover, the Service-oriented architectures paradigm allows for encapsulation of multiple self-contained, atomic services into a superordinate complex service with a service interface that can be invoked as well. The only prerequisite is that the interactions among the encapsulated atomic services are well defined. Naturally, the superordinate, complex

services can be further recursively encapsulated by another even more complex, superordinate service and so on.

In order to define interactions among SOA services, in terms of high-level application functional and temporal dependencies, *workflows* [AHK+02] present themselves as a useful tool. A workflow definition describes the prerequisites, the conditions and the plan of the necessary SOA service invocations, either atomic or complex, in order to meet a high-level application functional goal. In turn, the workflow definition plan is read by a workflow execution engine and is materialized into a workflow instance that is to represent the high-level application. The workflow engine carries out the system functionality in terms of workflow instance execution, by centrally managing all of its application services in a request-response fashion. Precisely, the engine's dedicated orchestration service invokes at runtime the specified application service, receives a response from it, and schedules the next application service for invocation provided with the outcome data of the previous invocation in a step-by-step fashion. This process is repeated till all integral application services of the workflow definition have been successfully invoked.

The introduction of workflow definitions necessitates the explicit distinction between the services, in terms of their type and role in the context in workflow instance execution. Precisely, we distinguish between application services and system services:

- *Application services*. Those correspond to integral functionalities of a high-level application that are encompassed by some workflow definition. Nodes that are hosting application services embody the SOA service providers and thus are subject to invocation by a workflow instance execution engine. Application services are usually provided by third parties.

- *System services*. Those correspond to the core functionalities of a workflow execution engine that is charge of managing (i.e., executing) the invocations of application services. An example of a system service would be the workflow instance orchestration service. System service enabled nodes embody the workflow instance execution engine.

Example 1.1

Figure 1.1 illustrates an example workflow definition that is drawn from a high-level weather forecast application scenario. As the figure shows the sample workflow is composed of a set of five activities, such as the data entry activity, the weather data retrieval activity, the map retrieval activity, the forecast computation activity and the centralized orchestration activity. Each of those activities corresponds to a SOA service of different type that is hosted at some computational node. Since each of those activities is different in their semantics it is correspondingly distinguished by a different geometrical shape (i.e., triangle, square, diamond and hexagon) and also maps to a different service (i.e., illustrated by the orchestrator, cog, weather and globe icons respectively). Moreover, the individual services differ in their roles in terms of workflow instance execution functionality. While the data entry, weather data retrieval, map retrieval and forecast computation activities map to application services and adhere to the weather

forecast application functionality the orchestrator activity maps to a system service and adheres to workflow engine functionality. The functional dependencies among the application services are depicted with red thick arrows, representing with the direction of the arrow the followed-by dependency. For instance, the data input service is followed by the weather data retrieval service. Note, that the data input service is also followed by the geographical map retrieval service at the same time and thus can be invoked in parallel once the input data service is finished. In turn, the forecast computation service, is dependent on the results of both mentioned parallel services and thus its invocation has to wait until both are finished. The functional dependencies among application services and the system service, i.e., the runtime invocations that are subject to enforcement by the centralized orchestration service are depicted with the numbered thin black lines. Thereby, the monotonically increasing numbers on top of the black lines correspond to the invocation order of the application services by the orchestration service. The parallel service can be invoked at the same time, hence they posses the same invocation order number. Finally, the shown example workflow serves only for illustration purposes and shall be further discussed, in terms of runtime execution details and associated problems in the subsequent chapters.

1.1 SOA Execution Environments

Nothing brings the trend of loosely coupled service integration on top of commodity hardware more to fruition than the model of *Cloud computing* [AG10]. In Cloud computing important actors in industry of computing make it their business to offer software services and excess resources, in terms of computation and storage, to end-users for (possibly commercial) utilization. Moreover, the Cloud promises their adopters resource quantities that are sheer unlimited on a pay-as-you-go model at affordable prices. Monetary flexibility and unprecedented resource availability incites the end-users to deploy ever increasing amounts of the most diverse applications into the Cloud which inevitability pushes their integration even further into more complex and coherent entities. Given the resulting high degree of diversity of end-user applications of the Cloud, integration can only be achieved by means of cross-platform, open-standard and loosely coupled integration means such as Service-oriented architectures. In fact, the services offered by the Cloud providers themselves are based on SOA so as to reach the biggest possible spectrum of users.

All offered services, such as storage, messaging etc. and computation XaaS (i.e, X as a service) are predominantly powered by commodity hardware. In order to provide the guaranteed availability of the services at any time, even in the presence of overwhelming end-user workload, Cloud computing necessitates gradual computational cluster expansion by adding new commodity devices to it. Thereby, the newly added devices can span the widest possible geographical regions so as to improve responsiveness to the end-user and do not have to be located within the same physical cluster. Moreover, the new devices usually feature upgraded performance characteristics, in terms of hard-

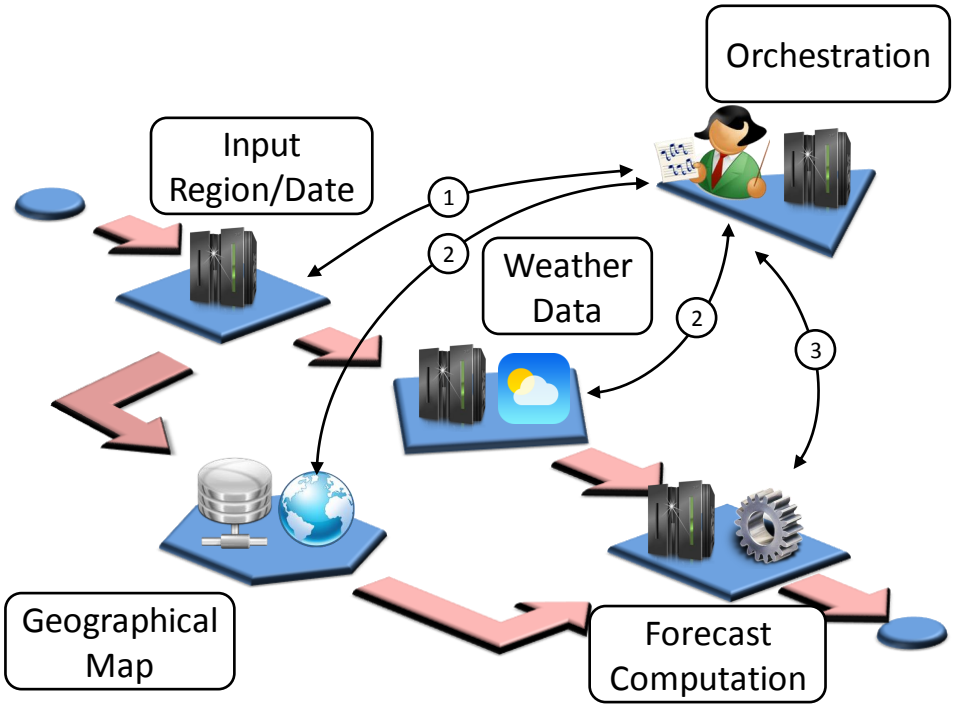


Figure 1.1: Workflow execution

ware, and can possibly be equipped with new software features that rely on SOA based integration with legacy software and applications.

Since Cloud computing merely stands for SOA enclosed and publicly available computational resources that are advertised by means of the vague *Cloud* term, the spectrum of its manifestations can be very broad. The extent of it can encompass thereby everything from small public computer clusters to huge privately owned data centers. For instance, academic institutes may provide public clusters for research purposes whereas commercial Cloud providers[1] may offer their data centers for commercial utilization purposes.

Form an end-user point of view, the CaaS (i.e., Computation as a Service) enclosed resources are offered in the form of prefabricated virtualized computational devices. Virtual computational devices emulate real ones with the benefit of being instantiable on-demand to theoretically unlimited numbers. This means that the end-user is agnostic to the underlying computational infrastructure and has to deal only with its own virtual devices. This allows them to focus only on things that matter such as application development. Moreover, by deploying the SOA backed applications onto the virtual devices new instances of them can be instantiated on demand, increasing thereby the application service quantities and thus availability. In practice, scalability of virtual devices, as it is called in Cloud computing terms, is narrowed down by the fixed price tag which is always associated to each virtual instance. Note, that Cloud computing usually implies many more resources as a service such as storage, messaging – XaaS – etc., however we limit ourselves in our work only to virtual devices.

[1] Amazon Web Services, Google App Engine, Microsoft Azure etc.

The result of the SOA fostered application deployments to the Cloud and their consequent integrations is a high degree application diversity, which might imply the most complex interaction scenarios. For instance, complex application interaction scenarios might even span devices which are outside of the scope of the controlled execution environment, such as the Cloud. That is, some applications might even rely on data or computations which are stemming from external devices, such as laptops, mobile smartphones, wearable sensors etc. The benefit behind SOA is that external devices can be seamlessly integrated into the Cloud based applications if they adhere to the SOA paradigm by offering computational or data resources as services.

Hence, Service-oriented architectures play an important role in bringing applications together, both for end-users as well as for Cloud computing providers. In the former case SOA enables application evolution by integrating diverse applications. In the latter case SOA enables application proliferation by providing the necessary means. By the same token, workflow orchestration can become a service of the Cloud itself, which subsumes all the necessary functionalities so as to manage the execution of workflow instances. This is in particular applied in the business domain, where it is referred to as Business Process Management as a Service (BPaaS) [SJV+15], and it enables the companies to dynamically expand their business models on a cost efficient and pay-per-use pricing model.

The consequence of applying SOA however is, that high-level applications usually incur a wide spectrum of computational devices, in terms of their hardware characteristics and hosted services, on the system that is charge of managing their execution, i.e., the workflow engine.

1.2 Distributed Workflow Execution

The spectrum of computational devices, a workflow engine has to face at runtime, can range from a potentially unlimited number of Cloud virtual devices to very few commodity and mobile ones. Thereby, the heterogeneity of the computational devices is usually abstracted from the workflow engine behind the services that are hosted at them. Service level agreements (SLAs), abstract representations of the service execution characteristics, usually reflect the host's hardware performance characteristics. Aggregated at the workflow engine, SLAs enable it to select services over others at orchestration time.

In the long run however, the centralized management of a virtually unlimited number of services at the workflow engine, hosted at a virtually unlimited number of heterogeneous computational devices (e.g., such as in the Cloud) for a virtually unlimited number of high-level applications puts the workflow engine into performance and reliability issues. In other words, a centralized workflow engine tends to become a *performance bottleneck* and *singe-point-of-failure* [Doo09]. Therefore, novel concepts to workflow management that addresses device heterogeneity in a salable fashion have to be introduced in the context of SOA based Cloud applications.

Workflow execution that stands out with high performance characteristics, in terms of the number of workflow instances it can concurrently run, inevitably entails distri-

bution concepts. If all specified application services of a workflow definition are redundantly distributed to the biggest possible extent of computational devices the system performance should benefit as a whole. By offering application services of the same type at more than one computational node, load balancing among them can be facilitated, by directing the invocations towards the best providers, in terms of workload utilization, available resources, physical proximity etc. As consequence, workflow execution instances should be optimally distributed among the service providers and system performance should increase. In the context of Cloud computing service distribution is effortlessly facilitated as new devices with the necessary service type can easily be added due to the unlimited resources of the Cloud platform.

Although application services are intrinsically distributed this is not always necessarily the case for workflow engine system services. The example workflow shown in Figure 1.1 highlights the problems of the centralized workflow management approach. As Figure 1.1 depicts the orchestration system service is centrally located only on one device and is thus always involved in all steps (i.e., numbered thin black arrows) of the workflow execution. On the other hand, all the other nodes participate to the execution of the workflows only when needed, that is only when their locally available application service is actually invoked. In case the orchestration service is confronted with numerous workflow instances and numerous application service providers to keep track of, its capability to carry out application service invocations will solely depend on its underlying hardware (i.e., CPU, bandwidth and memory) characteristics. Overloaded devices, in terms of workflow instances, offering the orchestration service will have to queue excess workflow execution enactment requests. As a result the performance of the system should degrade as a whole independent of the distribution degree of the application services.

The distribution of system services is only possible if it is accompanied with efficient management of data. In the course of a workflow definition execution among the encompassed distributed application services, data have to be exchanged. The results of one application service invocation are used as the input of another service instance invocation and so on. In the traditional sense of distributed workflow definition execution (Figure 1.1) all data exchange among application services has to go through the orchestrator node. At runtime, the orchestrator invokes an application service, collects the resulting data and exploits it for the subsequent application service invocation. As a consequence, all execution related data has to be stored intermediately at the orchestrator node itself. However, the more workflow definitions there are running in the system the more data associated to it exists that needs to be managed. Given the increase of concurrently running workflow definition instances, efficient means of intermediate data storage, in terms of access times, are essential. The most common way of achieving data access efficiency is of course by means of distribution of data among environment nodes.

Therefore, by employing the concept of a high degree distribution to services – both application and system – and data, produced as a result of their interaction, a high performance SOA based workflow management can be expected.

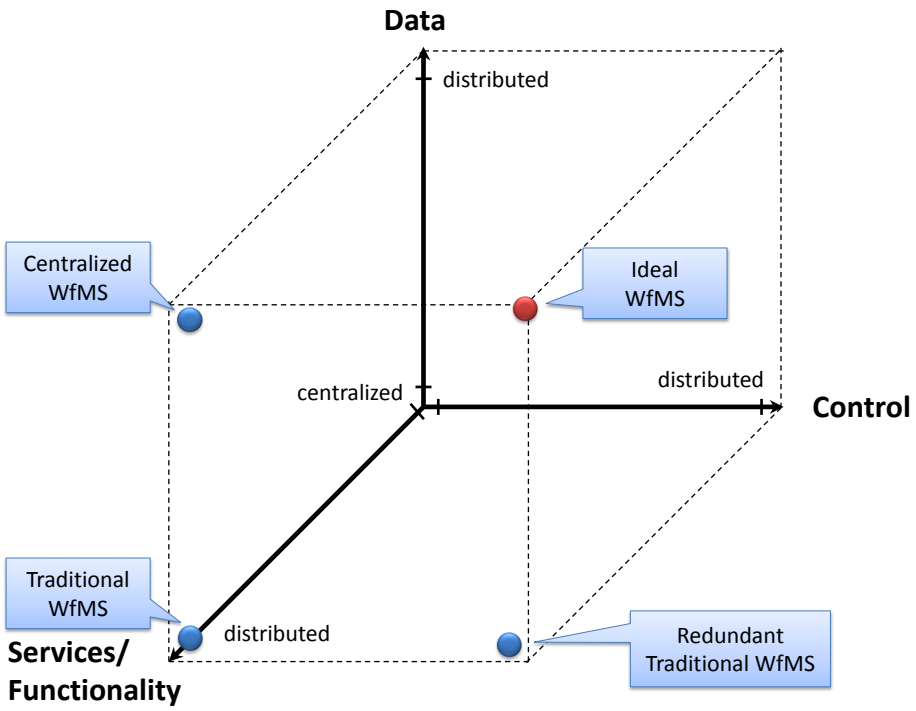


Figure 1.2: Workflow engine distribution aspects.

1.3 Problem Statement

Given the three aspects of distribution to workflow engines, namely application service, execution control, and data, distribution of it can be in a traditional sense facilitated in two ways. Either redundant and dedicated system service nodes are introduced into the environment or already existing application service providers are partially equipped with workflow engine functionality. With respect to the distribution aspects we distinguish between four different classes of workflow engines as follows: *traditional* engines, *centralized* engines and *redundant* engines. Figure 1.2 illustrates the traditional approaches to workflow engine distribution in terms of the three distribution aspects.

- First as a reference point we introduce the traditional workflow management systems (Enterprise service bus - ESB [Cha04]). Such systems feature simple distribution concepts by encompassing the invocation of distributed application services at the price of centralized execution control and corresponding data management. Moreover, local distribution concepts, in terms of the number of orchestrator nodes. Overload, in terms of numerous concurrent workflow definition instances, are addressed with multiple instances of the orchestration service at the same centralized node. The redundant instantiation of system services across the same node is referred to as *vertical distribution*. As consequence, bottleneck performance characteristics are exhibited eventually. To alleviate the bottleneck characteristics either the control over or the data over the execution is distributed from the traditional workflow management systems.

- Centralized engines [LYE+11] feature next to vertical distribution of system services also the distribution of data among nodes. Precisely, data management functionalities are outsourced from the orchestrator nodes to scalable distributed systems such as distributed databases or distributed file systems. Distributed data management systems feature a redundant instantiation of system services, in terms of data management, across a set of nodes which is referred to as *horizontal distribution*. Preferably, the distributed data management systems functionalities are featured by the application service hosts themselves. In turn, the orchestrator continue to centrally control the execution by instructing the application services to the data locations within the distributed data management systems. With this approach, bottleneck performance characteristics still prevail, however with respect to orchestration functionality of workflow definition instances only. Data management does not affect the performance any more.

- On the other hand, distribution of the execution control can also help to reduce the bottleneck characteristics of traditional systems. By horizontally distributing ([KBG+10]) the orchestration system service across a set of dedicated nodes the workload, in terms workflow definition instance orchestration, can be divided. To distribute all pending workflow instances among the redundant orchestrator nodes the most straightforward approach is to offer a global database that stores all pending instances. This database is queried by the available orchestrator nodes at runtime. Usually, the database has to be centrally managed so as to facilitate concurrency control among the orchestrator nodes. As a consequence, performance bottlenecks still prevail due to the centralized management of the database.

In the context of Cloud environments, applicability of established approaches to workflow management are questionable. On one hand, we have the centralized bottleneck characteristics. On the other hand, established approaches tend to incur additional costs on the end-user. Namely, each horizontally distributed system service node in the Cloud is associated with a fixed price tag independent of their runtime operating utilization. Suboptimal horizontal distribution results in higher cost eventually. For instance, in case of a high workload, in terms of concurrently running workflow instances, in system, additional nodes have to be spawned by the Cloud so as to accommodate it. Whenever this workload drops the additionally introduced system service nodes become underutilized. Hence, unnecessary costs are burdened on the end-users since the reduced workload can be effectively managed with fewer orchestrators. To overcome this problem, system service distribution needs to be powered by sophisticated and utilization-based allocation strategies. Optimal system service allocation strategies can only be applied by a centralized entity, i.e., a load-balancer. Hence, centrality of the established approaches is further increased and thus its inapplicability to scalable domains.

Example 1.2

Figure 1.3 illustrates such an example of the traditional orchestrator node service invocation. Precisely, Figure 1.3 shows a traditional load balancing enabled node (i.e., node

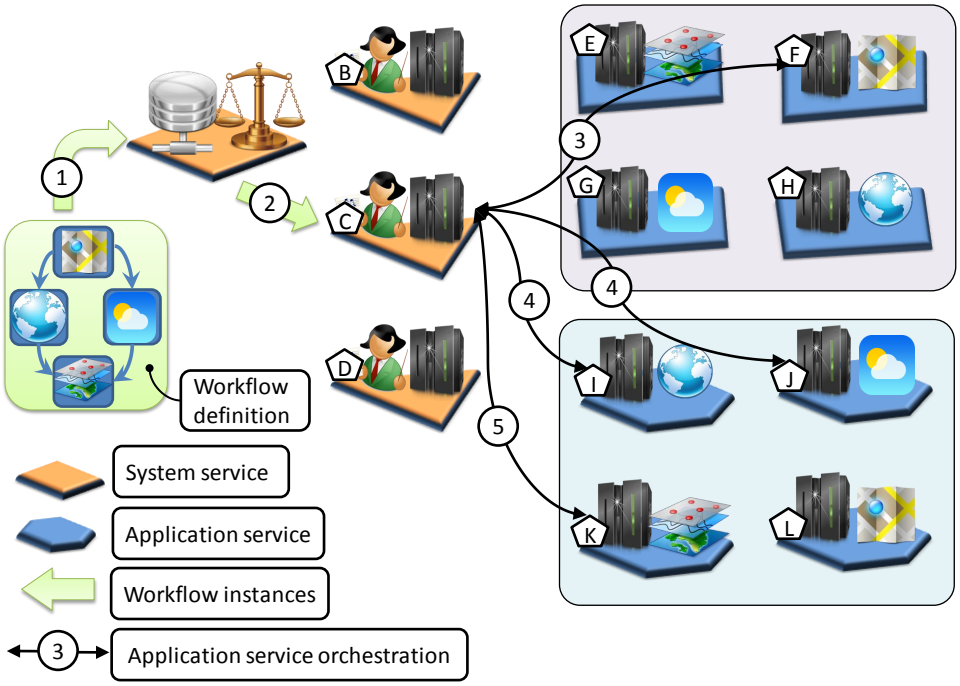


Figure 1.3: Traditional approaches to orchestration of application service invocations

id[2] A), three redundantly dedicated orchestration system service enabled nodes (i.e., node ids B through D) and two distinct clusters (i.e., node ids E through H for the first cluster and node ids I through L for the second cluster) of application service hosting nodes. The interaction order among the various participants of this setting is depicted with thin black lines and with thick green lines. Thereby, the thin black lines correspond to the order of application service invocations whereas the thick green lines correspond to the order of orchestrator workflow instance assignments. The exact interaction sequence is depicted with a lettered white circle. Moreover, the workflow definition is depicted that consists of four activities, i.e., a heat map computation activity (the three layer icon), a map retrieval activity (the globe icon), the weather forecast activity (the cloud and sun icon) and the coordinates location activity (the blue dot on the map icon). The activities themselves are organized into four invocation steps. As the picture shows a workflow definition is inserted into the load-balancer whose instance is immediately forwarded in a load-balanced fashion to the orchestrator C for execution. The orchestrator enacts the execution of the workflow instance by first invoking the application service (i.e., step 3) at node F. Once node F has finished serving the invocation request the services at nodes I and J are invoked (i.e., step 4) in parallel. Finally, the last remaining step of the workflow definition is executed by invoking (i.e., step 5) the service at node K. In the meantime, the orchestrator nodes B and D have not been utilized at all, thus creating unnecessary costs on the end-user.

[2]All nodes are distinctly represented with a node identifier letter that is enclosed within a white pentagon geometrical shape.

As Figure 1.2 suggests, the horizontal distribution of workflow definition management systems in the traditional sense is always subject to a trade-off between centrality of data or centrality of execution control. Given this centrality trade-off, bottlenecks should always prevail in case the traditional engines are faced with extreme scale workflow definition instance numbers. An engine that is to be considered *ideal* has to overcome this trade-off. That is an ideal workflow engine has to feature a high degree of distribution both in terms of data management and execution control. As Figure 1.2 suggests with the red dot, distributed data management and execution control functionalities have to be fused at the same time by the ideal engine. In other words, the redundant and dedicated orchestrator nodes have to additionally feature distributed data management functionalities. Thereby, the bigger the set of nodes that are equipped with the ideal workflow engine the better the performances should be, w.r.t. workflow definition instance numbers.

To reach the widest possible base of nodes for our ideal workflow engine the application service providers can be exploited. If the application service providers were to be enriched with workflow engine functionalities issues of centralized management could be effectively avoided. For starters, by vertically distributing system functionalities to them the resources of the service hosting nodes could be better utilized. For instance, workflow instance orchestration or data management can be performed while not serving application service invocation requests. In the context of the Cloud, this implies offloading of orchestration workload to underutilized existing service providers and thus saved costs, in terms of avoided additional node instantiations. Moreover, since the application service numbers of an environment are usually very high, the workflow engine would posses a wide base for the distribution of workflow definition instances even at extreme scale.

The enrichment of application service providers with fused system functionalities (i.e., orchestration and data storage) is however accompanied with increased node site management complexity, in terms of data management, reliability and service heterogeneity. Hence, novel concepts to distributed workflow management are necessary, in terms of an ideal engine.

Distributed Data Management

In order for any application service provider to be able of perform system services necessary metadata that powers them has to be available at all times. In the case of the orchestration service this is data on the execution environment, in terms of service providers, their hosted services, hardware characteristics, current load, physical proximity etc. Given the constantly changing characteristics of the underlying execution environment, in terms of existing devices, their available resources, their workload etc., the latest (i.e., correct) data on it has to be always accessible at runtime at all orchestrators. Stale and inaccessible metadata implies false orchestration decisions (e.g., service invocations on wrong nodes) and results in bad workflow instance execution performance. Moreover, orchestrators have to be provided only with the most necessary data if suboptimal resource consumption (i.e., storage) is to be avoided.

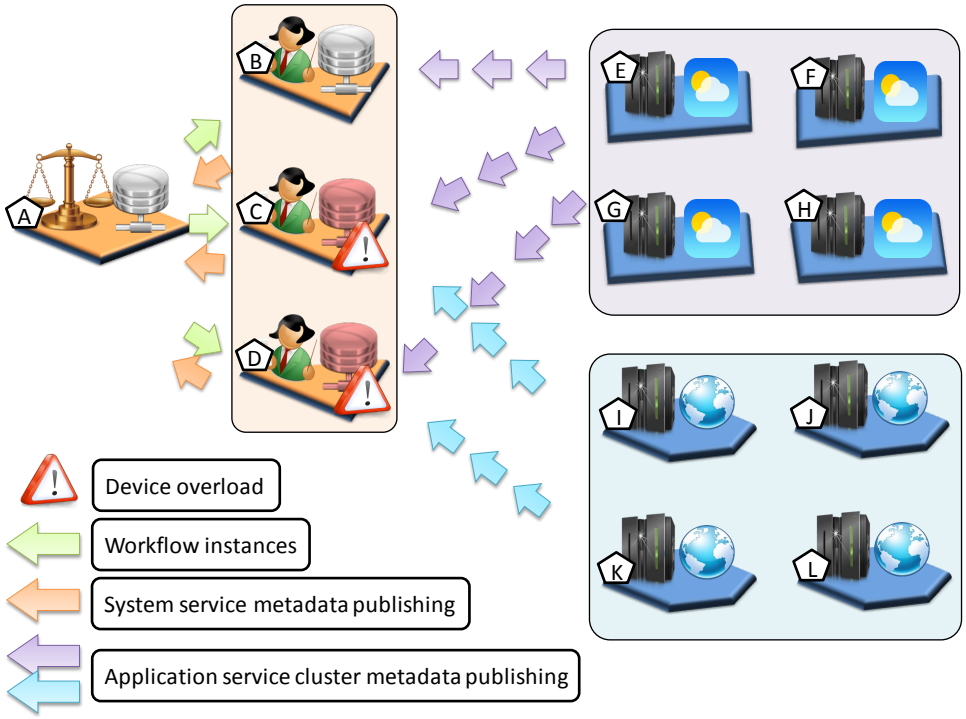


Figure 1.4: Traditional approaches of data propagation to service orchestration enabled nodes.

Traditional dedicated orchestrator node approaches only partially meet these requirements. By relying only on the load-balancer to interact with other participants, dedicated orchestrator nodes are inclined to aggregate metadata on the whole execution environment. This is not always feasible, in particular in the Cloud context. Due to unlimited resources of the Cloud vast numbers of application service providers can be spawned. Flooding the orchestrator nodes with data on all possible service providers of a scalable execution environment would result in early (e.g., storage) and unnecessary (e.g., CPU to manage all the data) resource depletion in the long run and cause a series of resulting problems. For instance, dedicated orchestrator service devices could not possess adequate local storage to manage all service providers, or devices could not possess adequate computational resources to process the incoming flood of data updates. As a consequence, orchestration nodes would be predominately occupied with managing data instead of orchestrating service invocations which implies overall bad workflow instance execution performance.

Example 1.3

Figure 1.4 illustrates an example of the traditional orchestrator node management of data. As in the previous example the same setting is show, in terms of the participating nodes and their hosted services. The only difference lies in the exact application services that are hosted, however this difference is not of significance to this example. Since orchestrator nodes are only observed in the context of data management they are not managing invocations of application services and are merely storing metadata (i.e.,

depicted with the database icon) on the execution environment. Likewise, application service hosts are only publishing their current state to the orchestrator nodes and are grouped together (i.e., depicted with the different colored boxes) with other devices of the same application service type. Moreover, the flow of metadata items, in terms of thick arrows, can be observed that is directed from the application service clusters towards the orchestrator nodes. Thereby, the amount of data (i.e., shown by number of arrows) and the origin of the data (i.e., shown by the cluster corresponding color) can differ. As the figure shows all orchestrator nodes are receiving metadata on node clusters, at which some are receiving more than the others. That is, the orchestrator nodes with the identifiers C and D are receiving data on both node clusters, whereas the orchestrator with the identifier A is only on one cluster since A is currently not aware of the other cluster. As the consequence, C and D are overloaded (reddish color of the database icon) with managing the incoming data whereas orchestrator with the id A is not

---

Therefore, distributing the orchestration service to devices that provide application service at the same time requires sophisticated data management concepts that guarantee freshness of data which allows for an optimal workflow instance orchestration.

Execution Environment Heterogeneity

Given the high degree of device heterogeneity that SOA application inevitably entail, deployments of services – both application and system – to them that are either random or static can additionally affect the distributed management of workflow instances. For instance, distribution of system services to external computational devices, such as for example in the context of complex interaction high-level application scenarios, which are characterized by instability and limited resources, might severely affect the execution performance of workflow instances. Orchestration performance of inappropriately deployed system services are likely to exhibit bad response times. For instance, is resource limited mobile devices are equipped with data/computationally intensive application services they will become overloaded with the service tasks and probably cause failures of their hosting devices due to excessive resource consumption and consequent early depletion. Such devices favor are rather suitable for services which are conservative on the local resources, and feature very limited functionality. For example, streaming services offer resource conservative utilization of local resources (i.e., in particular storage) and should be allowed on mobile devices only.

On the other hand, deployment of resource undemanding services to resource abundant devices, such as for example in the context of Cloud device instantiations, affects the execution of workflow instances as well. Orchestration decisions of such devices are likely to result in excellent response times, but at the expense of poor device utilization, which in the Cloud context results in unnecessary costs to their customers. That is, such devices are likely to spend most of the time underutilized (e.g., in idle state), creating additional expenses to their users as they are associated with a fixed price tag. Suboptimal service deployments manifest themselves particularly when huge numbers of corresponding workflow instances are run against them. Precisely, suboptimal ser-

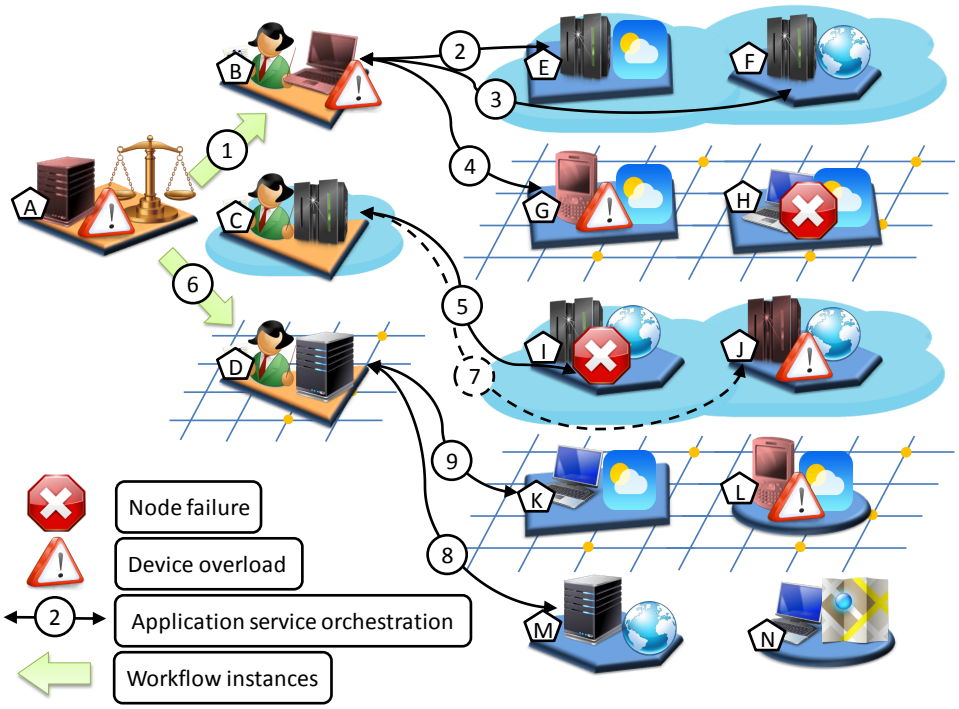


Figure 1.5: Traditional approaches of service deployments to heterogeneous environments

vice deployments perform like performance bottlenecks when a big number of service invocations are issued at them by queuing the excess ones.

Traditional approaches to service deployment at devices are based on static configurations. This implies, that services, independent on their type, are deployed statically only at the start-up of the device according to some high-level application strategy. At doing so the constantly changing characteristics of the underlying execution environment, in terms of available hardware, software resources, are completely disregarded. In the context of the Cloud, such an approach is not applicable as it reflect in unnecessary costs on the end-users. Static addition of on-demand services is usually based on instantiations of new hosting computational devices, which are naturally associated with prices.

Example 1.4

Figure 1.5 illustrates such an example of the traditional service distribution. As in the previous example the figure shows a traditional load-balancer (i.e., node id A), three dedicated orchestrator nodes (i.e., nodes B through D) and various application service providers (i.e., nodes E through N). However, unlike the previous two examples the execution environment of this one is slightly more complex, in terms of the node execution environment heterogeneity. In this setting we have overall 14 nodes of different origin, different device types and different hosted application services. While nodes C, E, F, I and J are devices which are located in some third party Cloud, the nodes D, G, H, K and L are nodes that are located inside some external grid. Nodes A, B, M and N are located

in some external execution environment. In terms of device types nodes C, E, F, I and J feature powerful mainframes. Nodes A, D and M feature high-end stationary desktop devices, whereas nodes B, H and K standard mobile laptop devices. Finally, nodes G and L feature low-end mobile devices. The application services are distributed among the nodes manually. The interaction order among the various participants of this setting is depicted with thin black lines and with thick green lines that can be also dashed. The semantics of the lines are the same as in Figure 1.3. The dashed versions of the thin black lines as well as the thick green lines correspond to failure recovery or load balancing actions, respectively. However, as we can see in this figure, nodes that are featuring low performance devices are overloaded in case they are hosting resource demanding services. These kind of situations may happen due to manual deployment of services by unaware (untrained) individuals, in terms of expected execution workload characteristics. For instance, the node B features an external laptop that has been manually equipped with the orchestration system service. Since the workflow instance, that has been assigned to it by the load-balancer, requires multiple service invocations (i.e., interactions $2, 3$ and 4) it is quickly brought to its limits as the laptop only features low performance characteristics. Likewise, the load-balancer (i.e., node A) is overloaded itself due to its featured desktop device that is not powerful enough to manage all orchestrator nodes and all workflow instances on time. As a consequence, the inadequately equipped nodes tend to become performance bottlenecks, hence the red color of their corresponding computational devices. In case of the node B load balancing is performed by sending a new workflow instance (i.e., interaction 6) to another orchestrator node.

That is why distributed workflow management has to take the dynamic heterogeneity of devices into account while introducing services to it. Service deployments should be part of a constantly ongoing and dynamic process that moves highly requested services to already existing resource abundant devices and vice-versa. This way execution bottlenecks, unnecessary costs, potential failures can be avoided and thus the overall system performance, in terms of running workflow instances, improved.

Reliability

The application of high degree service distribution in environments of very high computational devices numbers can be affected at runtime by failures. No matter how reliable individually, bigger computational environments eventually exceed the mean time between failures [3]. As a result, service invocations and orchestrations are likely to fail due to individual computational device crashes or outages. Moreover, if the computational environment is additionally extended to heterogeneous environments that contain mobile, wearable and resource limited devices failures are much more likely. Such devices may suffer from external damage or get out communication range in face of mobility. Given the runtime dynamics heterogeneous environments, static deployments of service types are likely to cause overload at some mobile devices and bring them to failure caused by resource depletion.

[3] Mean time between failures (MTBF) is the predicted elapsed time between inherent failures of a system during operation [Jon87]

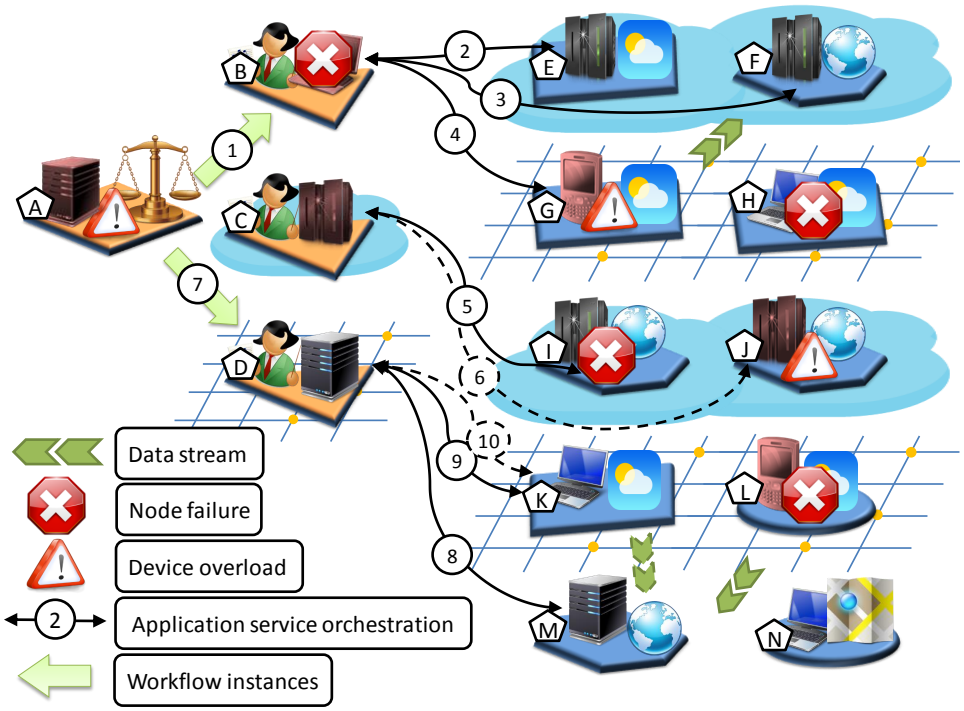


Figure 1.6: Traditional approaches to recovery of failed system and application services nodes.

In the event of node failures, the severity of the effects may differ depending of current context of the failed device. In case the failed device was only serving invocation requests from the orchestrator nodes at the time of the failure the significance on the overall workflow execution is not that high. Failed service hosts can be easily detected by orchestrator nodes by means of simple failure detection algorithms (e.g., periodic heartbeats) and replaced with other service providers of the same type for future reference.

On the other hand, if the failed device was enacting an orchestrator at the same time the consequences on the execution of workflows are far more severe. In such scenarios, the workflow instances currently managed by the failed orchestrator cease to exist as their execution state located at the orchestrator is lost along with it. In general, the recovery of orchestration nodes is far more challenging, as the lost workflow instances have be restored to the exact moment of failure. This is in particular difficult, if parallel invocations of one workflow stemming from the failed orchestrator have to be joined. In such situations all service providers that have been invoked in parallel have to join their results on one substitution orchestrator. The difficulty thereby lies in unanimously determining the most appropriate one out of a potentially huge collection of them in instant time.

The difficulties also apply to failed devices that are serving long running service invocation, such as streaming services, and thus feature an internal state. Also for stateful services, the lost state has to be restored to the exact moment of failure. This implies that along with state, the streaming data elements that caused it have to be restored as well.

Even if a single data element is lost, the lost state might never be recovered. The challenge behind the recovery of stateful services lies in the fact that storing a continuous flow of data is physically not feasible. Especially the resource limited (mobile) devices can not afford to store a continuous flow of data. The state and selected parts of the continuous data flow could be stored at the orchestrator nodes but they are subject to failure themselves.

Traditional approaches to application service provider and orchestrator recovery focus solely on individual service role failures (e.g., application service provider failures) and entail significant overheads, in terms of data storage or computation (i.e., service) redundancy. In the presence of device failures established fault-tolerance mechanisms rely on redundancy of data that reflects workflow execution state or application service state so as to guarantee system availability. The redundancy of data is carried out with timely replication to stable backup sites. The backup sites are either tightly coupled to the orchestrator or organized at devices that feature the same services as well. In any case the assumption is made that the redundant backup sites are *stable* and thus impervious to failure. However, this assumption cannot hold in scalable environments, due to the mean time between failures of nodes, or in heterogeneous environments, that are composed of resource limited devices. In the long run, the approach of excessive data redundancy is associated with scalability problems w.r.t. the number of running workflow instances. In case the orchestrator serve as backup sites, the computational and storage effort of redundantly preserving huge quantities of workflow instance state reduces their capacity to process service invocations and thus system performance. In case the resource limited service providers serve as backup site themselves, excessive replication overhead reduces the amount of available resources so as to conduct long running execution. On the other hand, if inadequate data redundancy is provided, the system might not be capable of recovering from failures by substituting the failed invocations when the number of failures outweighs the number of replicas.

Example 1.5

Figure 1.6 illustrates such an example of the traditional node failure recovery mechanism. In this example the same execution environment setting as in Figure 1.5 is used, i.e., the traditional load-balancer (i.e., node A), three dedicated orchestrators (i.e., nodes B through D) and the cluster of heterogeneous nodes $E - N$ of application service providers are shown. Since orchestrator nodes for this example are only observed in the context of workflow instance execution failure-recovery they are managing invocations of application services. Much like in the previous example the interaction among the nodes, in terms of invocation orchestration and workflow instance load balancing, is displayed in the same way by means of thin black lines and thick green lines. The dashed versions of lines correspond to failure recovery actions. In contrast to Figure 1.5, we assume here that the weather application service maintains state and thus streams data.

As this figure shows the orchestrator at node C continuous the orchestration of the workflow instance, once the original orchestrator at node B fails. That is, orchestrator C resumes the invocation at step 5 by invoking the map service at node I. To this end the orchestrator C has to share the execution state of orchestrator at B at all times. The

subsequent failure of the service at node I can be recovered by the orchestrator (i.e., C) at a node of the same hosted service type – node J at dashed step 6. The workflow execution state is located as a backup at it, thus it is capable of doing so. In case orchestrator C becomes overloaded due to overall extensive backup state management, the load balancer has to redirect the workflow execution to node D (i.e., at step 7) which orchestrates the other steps and resumes recovery responsibilities. For example, at step 10 when the streaming service at node L fails.

As we can see from this example, an orchestrator has to maintain an abundance of data, in terms of workflow execution state and streaming services state, so as to be able to recover all possible failure scenarios. This state has to be shared (i.e., replicated) with the other orchestrator nodes but also with the load balancer. This tends to overload it in face of voluminous data in the long run. In turn, the load balancer has to be additionally outfitted with orchestrator recovery functionality which further increases its bottleneck (hence the red color) and single-point-of-failure characteristics.

Therefore, novel concepts to fault-tolerance are needed that can cope with the ever increasing number of running workflow instances. Thereby, is should seek to provide a high degree of robustness but not at the price of excessive resource consumption and performance degradation. An ideal solution should exploit the good characteristics of the underlying execution environment such as abundance of backup nodes candidates so as to distribute the recovery overhead.

1.4 Thesis Contributions

To address the challenges of distributed workflow management as discussed in the previous section this thesis offers the following contributions:

- A formal system model that describes the execution of workflows in a distributed setting. The model defines workflows w.r.t. to their structure and their runtime behavior. It captures the constitutional parts of a workflow into discrete and continuous services. Moreover, the model defines system services that manage the distributed execution of workflows and considers the distributed environments for which the system services are applicable.

- A novel concept to reliable distributed workflow execution that tackles the problem of node failure recovery in a scalable fashion. In particular the Safety-Ring system service is provided that offers failure-handling for a wide range of node failures, which host active discrete services of running workflows. The Safety-Ring service offers self-organization and self-healing features such that it provides reliability for itself while supporting scalable execution of workflows at the same time. The Safety-Ring service is implemented on top of the OSIRIS distributed workflow engine and its impact validated with empirical performance evaluations.

- An efficient data access protocol for the Safety-Ring system service – Compass. The Compass protocol aims at optimizing for the latency among nodes while providing access to data in the Safety-Ring. Combined with Compass, the Safety-Ring service is made applicable even for the most heterogeneous of node environments. The Compass protocol is implemented on top of the Safety-Ring and its impact validated with empirical performance evaluations.

- An enhanced recovery protocol for continuous service types – redundant passive-standby. Building on top of existing recovery mechanisms, such as passive-standby, it increases the redundancy of service state so as to improve the robustness of continuous services to node failures. In doing so the redundant state is enforced with a data consistency protocol such that correct recovery can be provided at all times. The data consistency protocol is lightweight, in terms of bandwidth overhead, which allows its application to resource limited node environments. The redundant passive-standby protocol is implemented on top of the OSIRIS distributed workflow engine and its impact validated with empirical performance evaluations.

- A novel concept to distributed workflow execution optimization that tackles the problem of static service distribution. At the heart of our approach lie decentralized controllers that autonomously perform dynamic deployments of services with the goal of improving the throughput of workflows in the system. Thereby, the controllers focus on application services and the Safety-Ring system service. The dynamic deployment of application services is implemented on top of the OSIRIS distributed workflow engine and its impact validated with empirical performance evaluations.

1.5 Thesis Outline

The composition of this thesis consists of nine chapters. The first two chapters, i.e., this chapter and Chapter 2, focus on introducing the research field of workflows. In doing so Chapter 1 and 2 discuss the difficulties of workflow management in distributed settings and elicit the contributions of this thesis. A detailed elaboration of the difficulties w.r.t. distributed workflow execution are presented in Chapter 2 where the workflows are applied to a highly dynamic heterogeneous environment, in the context of modern emergency management scenario. Chapter 3 provides detailed discussion of fundamental distributed data management concepts of which this thesis is based. Chapter 4 introduces the formal system model in the context of distributed workflow execution. Chapter 5 covers the conceptual contributions of this thesis. It discusses the novel self-healing workflow execution model in terms of discrete and continuous service failures. It discusses the Safety-Ring system service, its Compass extension and redundant passive-standby. The self-optimizing workflow execution model that features dynamic deployment of services is discussed in this chapter as well. In Chapter 6 the implementation methods for the novel concepts are provided. The OSIRIS Peer-to-Peer workflow execution engine, which serves as our implementation basis is introduced in this

chapter as well. Chapter 7 describes and discusses the quantitative evaluations of our contributions, whereas qualitative comparisons to other relevant approaches are drawn in Chapter 8. Finally, chapter 9 concludes this thesis and provides an outlook into the future.

2

Motivation

In this chapter, we describe an example application scenario that illustrates the challenges on a modern workflow engine. The application scenario we have chosen stems from the field of modern disaster management. We provide an example workflow that is composed of SOA backed services and highlight the issues associated with the management of its instances. Finally, we summarize this chapter with a list of requirements on a modern workflow engine.

2.1 Use Case

To motivate the application of Service-oriented architectures that is organized by means of workflows and powered by Cloud infrastructure consider the following firefighter rescue scenario. In this scenario we have a group of firefighters that are sent out into the field with the task of extinguishing a fire. Usually, this group of firefighters is subdivided into smaller units and strategically directed along with their firetrucks into different geographical parts of the action area with preassigned specific tasks so as to contain the fire from all possible sides. Given the nature of their jobs, triggered by various environmental and human causes, critical situations can occur that put the firefighters lives into risk. For instance, a sudden change of the wind direction can make the fire spread in an unexpected way that has the potential to put the firefighters into danger by entrapping them. Due to the firemen's individual and limited perception and view of the overall scene, also caused by human factors such as fear, fatigue etc., they most probably cannot anticipate such situations. In order to reduce the risk in a traditional sense, firefighters rely on mutual coordination among themselves and with the headquarters, by means of bulky and outdated communication devices such as radios. However, given the high dynamics of their tasks, which often includes hard physical labor, it is not always possible for them to communicate with each others. Moreover, radio devices might get damaged due to external causes throughout the course of an intervention, and thus potentially lifesaving information is not shared. As a consequence, the firefighters coordination possibilities in the field are rather limited, and the risks for their lives high, if they are only based on simple radio devices. By applying the lat-

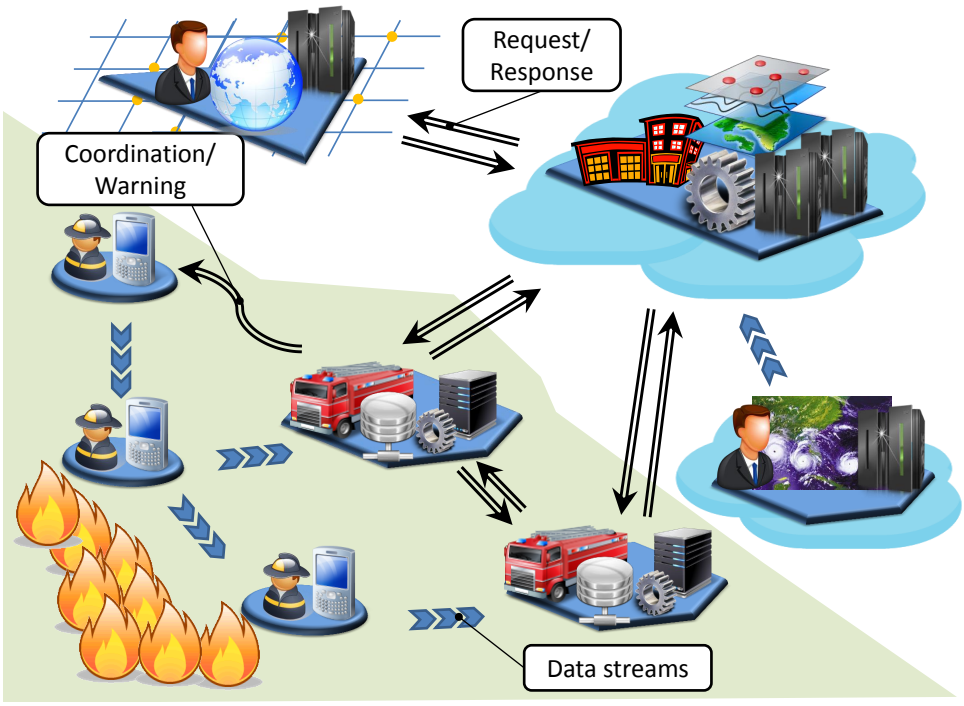


Figure 2.1: Modern disaster management

est technological advances in the fields of mobile communication devices, surveillance and intelligent decision support software their coordination effort could be immensely helped.

As a vision for the future, consider a wearable intervention support system for the firefighters. This system implies a variety of sensors that are spread out in the action area so as to capture as much data as possible during the course of an ongoing intervention. Among others, sensor data includes the physical state of the firefighters, their actions, the environmental state of the action area etc. In conjunction with other already existing data (e.g., weather forecasts) the data produced by the various sensors is subject to sophisticated analyses with the goal of anticipating critical situations of rather complex semantics. The outcome the data analysis process, which should ideally happen in near real-time, are danger area heat-maps. In turn, the danger heat-maps enable the firemen to avoid critical situations and coordinate better.

To produce the data that is needed for the heat-map creation process, the appropriate data producing and processing devices should be strategically distributed among all participants of the disaster management scenario. Regarding the firefighters, each of them should be equipped with a set of portable data capturing devices such as a camera, a GPS sensor, a accelerometer and other vital signs sensors that constantly produce (i.e., record) data on their individual actions. Moreover, a mobile computational device, such as a smartphone, should be carried as well by the firemen with the purpose of integrating and recording all data sources. The locally collected data at the firefighter smartphones should be subject to simple local processing, i.e., mainly filtering of the data sources, before it is sent away for further processing. Thereby, the locally collected

data should be transmitted – via wireless communication – to the nearest fire-truck. In case no fire-truck cannot be reached, due to distance or physical obstacles, data should be nevertheless transmitted via other firemen's devices that are in transmission range of any fire-truck.

The fire-trucks should be in turn equipped with more powerful computational devices (e.g., laptops, desktops) and more sophisticated environmental sensor devices (e.g., wind sensors) that are too heavy or too sensitive to be carried around by firemen. The aggregated data stemming from firefighter mobile devices should further be enriched with the additional sensor sources and shared with other fire-trucks in case there are any. By collaboratively performing simple data analysis already at this stage with all computational devices located at the firetrucks the firemen should be coordinated better. With the application of simple rules and databases that convey simple business logic at the fire-trucks immediate threats can be detected and preemptive warnings issued.

At the final stage of heat-map creation process, aggregated data at the fire-truck devices should be forwarded to the headquarters where it is subject to more sophisticated analysis with the usage of intelligent software agents. In doing so, powerful computational clusters, such as the Cloud, should be exploited so as to power the computationally and data intensive agents with excessive computational resources and voluminous data, i.e., resources the cannot be easily provided to the fire-trucks. In order to improve the quality of results, historical data and already existing data sources of third party external providers, such as weather forecasts, geographical maps etc., should be exploited as well. Based on the outcome of the data analysis process optimal coordination strategies should be devised – for future reference – and thus the risks for the firemen reduced.

2.2 Dynamic Execution Environment

A global overview of the aforementioned firefighter intervention support system is depicted in Figure 2.1. As the figure shows, the execution environment in this scenario is highly heterogeneous, in terms of the computational devices (i.e., their types) and application services that are participating to it. Regarding the computational devices, we can distinguish between four classes, namely the firefighter devices, firetruck devices, computational cluster devices. Their specific runtime execution characteristics can be defined as follows:

- Firefighter devices. The devices that are carried by the firefighters feature rather limited performance capabilities of very high dynamics due to the limited resources at disposal, in terms of CPU, storage capacity and bandwidth. They are further characterized by a rather high failure probability due to their limited lifespan (i.e., limited battery life) and the mobility of the firefighters. Due to the inherent movement of the firemen, they are more likely to disconnect unexpectedly (e.g., reachability) from the others, to have their transmission obstructed (e.g., obstacles), or to suffer externally induced damage by surrounding objects. As a consequence, they are not very suitable for computationally and data intensive tasks

or tasks that are critical to the overall intervention support system. Hence, their applicability should be restricted to simple tasks (e.g., monitoring of other mobile devices, data filtering etc.) and any conducted computation or produced data has to be immediately secured, by means of sophisticated fault-tolerance mechanisms, to more reliable sites so as to prevent losses. Since only a fixed amount of firefighter mobile devices (i.e., initially deployed ones) is to be expected throughout the course of intervention the most viable option that offers itself (next to other firefighter devices) are the fire-trucks.

- Fire-truck devices. Unlike the mobile devices of the previous category, fire-truck devices feature much higher performance characteristics. Since they do not have to be carried by firemen they can be equipped with more powerful hardware that corresponds to high-end desktop computers. This allows them to take on more challenging tasks of the intervention support system, such as data aggregation and simple analysis. By statically assigning them to fire-trucks in fixed numbers they are less likely to fail as compared to the mobile devices. Although they are static, they can nevertheless disconnect from the system (e.g., induced by environmental obstacles) due to physical locations of the fire-trucks which are outside in the field. That is why fire-truck devices necessitate reliable means of communication, in terms of hardware and software solutions, that guarantee lossless data transmission. Especially, since their main role is to consolidate the mobile devices with the remote computer clusters as junctions.

- Computational cluster devices. Devices of this category feature the best and the most reliable performance characteristics of all. Usually, they are part of a controlled and static execution environment and thus are unlikely to fail. In conjunction with a huge number of identical computational devices they can take on the most computationally and data intensive tasks, such as the heat-maps creating intelligent agents. However, these devices necessitate an intelligent distribution of tasks and data among a huge number of them as a key powering concept. In the context of the Cloud the number of computational devices can be dynamically scaled from very small to possibly infinite by dynamically adding and/or removing them from the system. By supporting such devices, with scalable distribution concepts so as to power the tasks at hand any kind of end-user performance requirements can be met.

The observed execution environment heterogeneity can only be handled by means of Service oriented architectures. Only if the functional roles of the devices, with respect to the intervention support system, are encapsulated behind SOA application services their mutual interaction is possible. Thereby, the data produced and the computations conducted by the services can differ in the quantity and frequency it is performed. Due to this, the way services can interact among each other can also vastly differ. Precisely, we distinguish between three classes of application services w.r.t. interaction type, namely the discrete services and streaming services Their specific interaction characteristics can be defined as follows:

- Discrete services. This class of services corresponds to the simple request-response interaction model. That is, services of this type are only invoked with an incoming request by a client that triggers the execution of underlying computations. Once the computations have been finished an outgoing response is sent to the very same client that issued the request. Thereby, the incoming request can be accompanied with data and likewise the outgoing response can contain data as the outcome of the underlying computations. The exact execution time of discrete invocation is depending on the computational intensity of the service. The invocation clients can be other services or applications of any kind that adhere to the SOA interface of the service. Moreover, discrete services can be stateless and/or stateful which means that they can maintain internal state between invocations. That is, stateless discrete services do not maintain internal state and always produce the same output (data) response for the same input (data) request. Stateful services maintain internal state based on which (i.e., its current value) the output responses are created. Services of discrete type are predominantly to be found at cluster computing devices as they are used for the encapsulation of long lasting complex computational tasks, such as heat-map creation.

- Streaming services. This class of services corresponds to the push/pull interaction model. Once invoked, services of this type continuously produce/consume output/input data items, based on the underlying computations of the service, for the client that invoked them. Invocations of such services can be achieved by means of discrete calls upon which they continue to periodically push/pull data till they are deactivated also by a discrete call. In order to be able to pull/push data items, invocation requests of streaming services have to specify the exact sources/destinations of the data. A source/destination of a service is again a service itself. By specifying the source streaming services can periodically pull data items from them for a longer period of time and process them according the underlying computations of the service. Once the pulled data items have been processed the produced data items are periodically pushed to their destination. The exact frequency of data pulling/pushing depends on the computational intensity of the service and end-user requirements (e.g., in case the frequency is too high it can be throttled down by the service). Similar to the discrete ones, streaming services can maintain internal state that results in different subsequent (data) outputs for same (data) inputs. Services of streaming type are predominantly to be found at mobile computing devices of the firemen as they are used for the encapsulation of tasks that feature long lasting data production and simple data processing.

Figure 2.2 depicts in a global overview the vast execution environment w.r.t. the heterogeneity of the intervention support system. As we can see the environment is mainly defined by three dimensions, namely the device quantity, the device types and service types that it encompasses. Thereby, the spectrum of device quantities can range from small to possibly infinite (e.g., Cloud) and strictly static to completely dynamic (e.g., Cloud). Regarding the device types, the spectrum can range from homogeneous (e.g., third party computer cluster) one type to heterogeneous multi type devices. Regarding

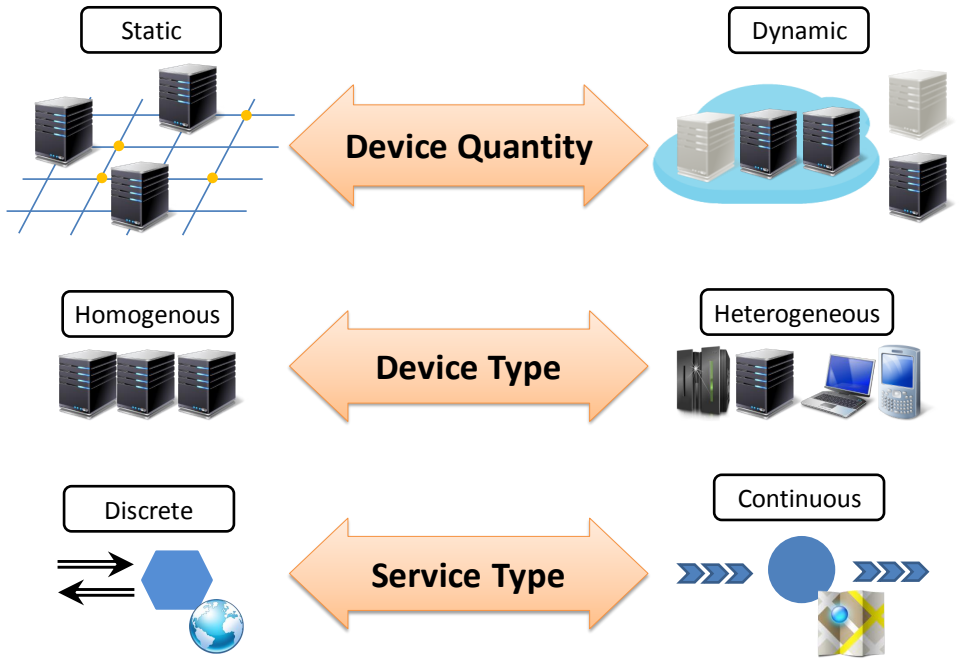


Figure 2.2: Heterogeneous execution environment of the application scenario

the service types, the spectrum can range from simple, request-response, and stateless discrete services to continuous, stateful streaming services.

2.3 Modern Disaster Management Workflow

In order to encompass the aforementioned application services, in the context of the intervention support system, workflows shall be used. Figure 2.3 illustrates an example workflow definition that covers the most important aspects of the disaster management application scenario. Precisely, the shown workflow definition represents the activities and their mutual interactions in *BPMN*[1] notation that are necessary so as to create the danger area heat-maps.

In a nutshell, the workflow definition specifies that in the first stage of it data from various sources has to be obtained in parallel, before the computations of the heat-map can commence. Once the data has been obtained and the heat-map computed it is used together in the second stage with other data sources to compute a coordination strategy. In the final stage the results of the first two stages are stored at the appropriate sites for future reference.

[1]Business Process Model and Notation (BPMN) is a graphical modeling notation that is handy for illustrations of flowchart diagrams of rather complex semantics, such as workflow definitions in our case. BPMN stands out with intuitiveness in representation which makes it applicable for a broad spectrum of users, and in particular for non-technical laypersons such as business users.

Each one of the workflow definition encompassed activities directly corresponds to a SOA service invocation. Each of the invoked services is represented with a separate *BPMN* swim lane. The figure displays six services that are numbered with the white boxes. All except for the service no.1, which is the workflow instance orchestration (i.e., system) service, are application services of some sort. Hence, only service no.1, i.e. the orchestration service is capable of reading the workflow definition and managing its execution. Thereby, the functional dependencies among the application services are depicted with thin black arrows, representing with the direction of the arrow the followed-by dependency. The dashed blue bi-directed arrow represents a request-response discrete invocation by the orchestration service, i.e., the functional dependency of the application services to the workflow instance execution engine. Details on the application services area as follows:

- Service no.2 – Data Aggregation service. Out of all application services, this service represents to a service of streaming type and is located at some fire-truck that is connected to the other services via a network. That is, service no.2 is in charge of collecting the streaming data from the various data sources and providing them to the disaster management workflow instance execution.

- Service no.3 – Strategy Service. This service is of discrete type that is computationally and data intensive. It provides data management and computations services for firemen coordination strategies. The strategy service is located on a device inside a Cloud execution environment.

- Service no.4 – Geographical Maps Service. This service is of discrete type that is data intensive. It provides third party data management services for geographical maps. The Geographical Maps service is located on a different device inside the same Cloud execution environment as the previous service.

- Service no.5 – Heat-Maps Service. This service is of discrete type that is computationally and data intensive. It provides data management and computation services for danger heat-maps. The heat-maps service is located on a different device inside the same Cloud execution environment as the previous services.

- Service no.6 – Weather Service. This service is of discrete type that is computationally and data intensive. It provides third party data management and computation services for weather forecasts. The weather service is located on a different device inside the same Cloud execution environment as the previous services.

Further details of the application service internal activities are omitted for simplicity reasons except for the case of service no.2.

Due to the overwhelming majority of discrete services the example workflow definition is primarily intended for an execution which is based on the Cloud execution environment. For instance, the workflow engine can periodically triggered the execution of this workflow so as to keep the heat-maps up-to-date during the course of an intervention.

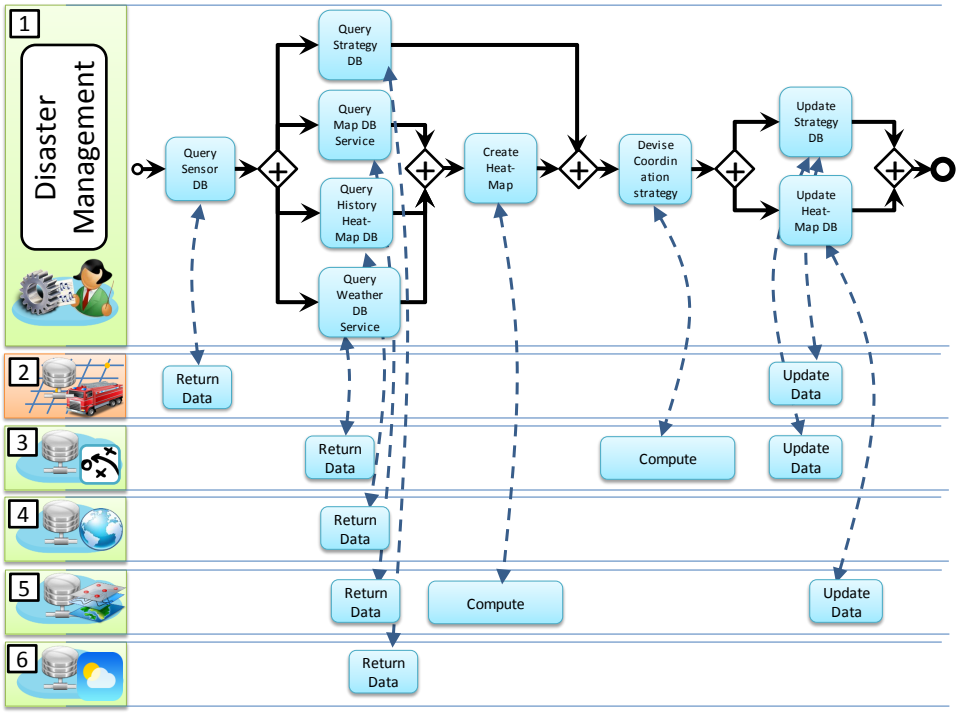


Figure 2.3: Example workflow definition of the application scenario

Given the modular nature of SOA the rather abstract service (i.e., service no.2) can be further broken down into simpler services and also represented with a workflow definition. Figure 2.4 illustrates the Data Aggregation service in *BPMN* notation as well.

In a nutshell, the workflow definition of Figure 2.4 specifies that in the first stage of it incoming data is read from data source and immediately stored to the internal database for aggregation purposes. Afterwards the filtering of the incoming data as well as retrieval of external data has to be performed in parallel before simple assessment of the incoming data can be performed in the second stage. In case data assessment has returned critical values the data source is informed by sending a message to it in the final stage and the whole process is started from the beginning, leading this way a service of streaming type.

Similar to the global Disaster Management workflow definition some of the activities correspond to services which are represented with the same BPMN notation, in terms of swimming lane and arrow meanings. However, in Figure 2.4 only four services are depicted at which service no.1 assumes also the workflow instance orchestration system service functionality. In the context of this workflow definition type, the system service is a gateway to all the other services and enforces the Data aggregation application service by orchestrating their invocations. All of services are co-located on the same fire-truck and together they comprise the Data aggregation application service that is accessible to other nodes via a network. The other depicted services (i.e., no. 2, 3 and 4) correspond to simple application services, whose details are as follows:

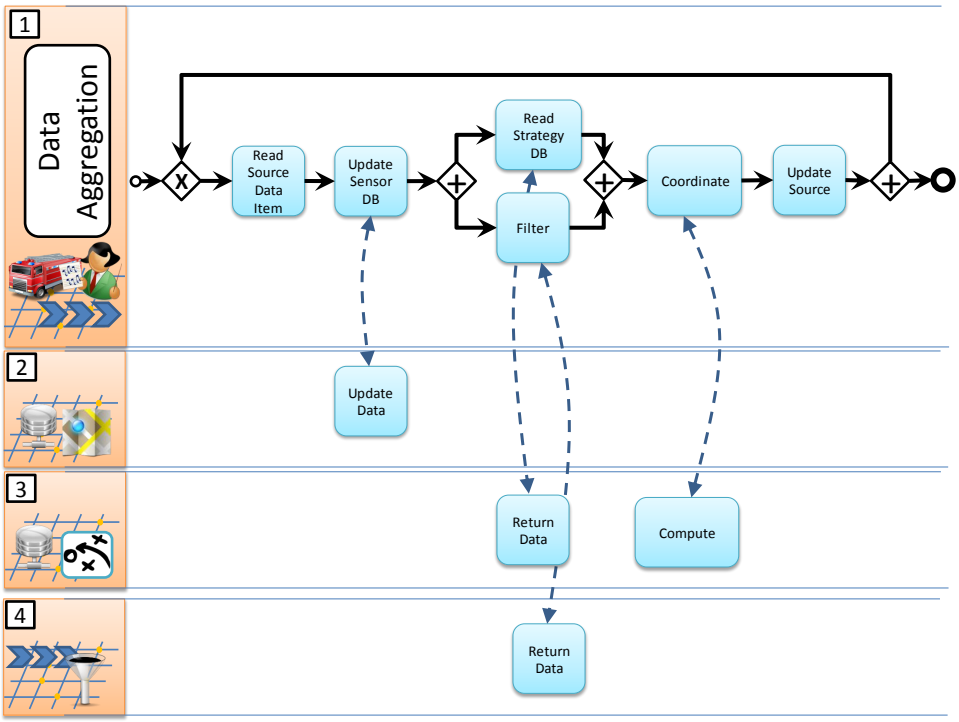


Figure 2.4: Data aggregation service

- Service no.2 – Sensor Data Service. This service is of discrete type that is data intensive. It provides local data management services for incoming sensor data. The sensor data service is located on the same device as the system service. The managed data of this service is publicly available for queries to other users, such as the Disaster Management workflow definition.

- Service no.3 – Strategy Service. This service is of discrete type that is data and computationally intensive. It provides local data and computation services for for firemen coordination strategies of smaller scale. The strategy service is located on the same device as the system service. The managed data of this service is publicly available for queries and updates to other users, such as the Disaster Management workflow definition.

- Service no.4 – Filter Service. This service is of streaming type that is computationally intensive. It provides local computation services for the cleaning of incoming data. The filtering service is located on the same device as the system service.

By encompassing discrete and streaming services and supporting their types of interaction within a workflow definition it appears from a higher level point of view (e.g., from a Disaster Management workflow definition point of view) as a streaming service itself, provided an encapsulating high level SOA service exists.

2.4 Workflow Engine Requirements

Given the heterogeneous execution environment of the disaster management application scenario, and its corresponding workflow definition, we summarize the following conceptual requirements that are put before a modern workflow execution engine, in particular its orchestration system service:

- Service compatibility. Given the all the different types of interaction among the services (e.g., discrete, streaming etc.), encompassed by the disaster management application scenario, the engine should provide means of communication that supports their combinations. Moreover, the most heterogeneous execution environments that can find themselves hosting SOA services should be supported as well.

- Reliability. Given the volatile conditions the firefighter mobile devices can find themselves in or the Cloud MTBF, the engine should feature fault-tolerance mechanisms that prevent the loss of workflow instances and its associated data due to the failure of devices. This fault-tolerance mechanism should be resistant to any kind of node failure (e.g., temporary, simultaneous, permanent device failure) and be feature good throughput performance even in the presence of many workflow instance numbers.

- Scalability. Prominent characteristics of the underlying execution environments, such as the Cloud scalability, should be exploited by the engine with advanced distribution concepts, in terms of data management and system service deployment. That is, the presence of a possibly infinite number of nodes to the execution environments and its associated management of data should not bring down the performance of workflow instance execution. Rather by strategically distributing data and system services across many nodes performance should be enhanced instead.

- Resource conservation. The workflow engine should feature an easily customizable and lightweight software stack, that stands out for its small hardware footprint, in terms of CPU, memory and bandwidth requirements. The stack should not claim too much local resources so that application services are affected. A small hardware footprint in conjunction to custom configurations of the engine should allow for deployments even to resource limited (i.e., mobile) devices.

3

Distributed Data Management

The purpose of this chapter is to offer insights on most fundamental distributed system concepts that have found a broad application in industry and research. We discuss in this chapter fundamental data management concepts, in terms of scalable data distribution, reliable data availability and data consistency, as they will lay the foundation for our work in the upcoming chapters.

In order to meet the requirements of an ideal workflow engine to the biggest possible extent, horizontal distribution of the engine across a number of nodes has been identified as the primary method. Although service oriented architectures imply the inherent distribution of application services within an environment, the execution engines' functionalities (e.g., orchestration and data management) had to be distributed in a traditional sense by selecting a dedicated and redundant set of nodes to host them entirely. By restricting the pervasive system functionalities to a few dedicated only nodes, numerous challenges arose, in terms of reliability, data management etc.

As Figure 1.2 suggests, the horizontal distribution of workflow engines in the traditional sense is always subject to a trade-off between centrality of data or centrality of execution control. This trade-off refrains the traditional workflow engines from application extreme scale workflow definition instance number domains. Ideal workflow engine however necessitate scalable horizontal distribution concepts without centrality compromises whatsoever.

One notable and uncompromising approach to extreme scale distribution, for any system functionality aspect, is featured by the *Peer-to-Peer* (P2P) [Ora01] networks. In Peer-to-Peer architectures all participating nodes of a network feature the same functionality stack that is utilized in close collaboration with other nodes without the need of a centralized coordination entity. Peer-to-Peer nodes collaboratively share the overall workload of a global functionality aspect (i.e., its burden) by individually and partially contributing to it. Thereby, peers put their hardware resources at disposal to run the functionality stack but only for a partial domain of the global functionality. The consequent partitioning of the functional domain among peers results in high scalability of peer numbers and thus a high performance of the global functionality stack.

In upcoming sections we take a deeper look into the peer-to-peer domain and elaborate techniques that are intended for the storage of huge quantities of data in a scal-

able, reliable and consistent fashion. The concepts to distributed data management presented in upcoming section will serve as the most important building blocks of our work as they will enable us to establish synergies among distributed data management and execution control system functionalities. From a distributed data management point of view, P2P concepts come into play by subjecting all nodes to collaborative data management burden sharing. The most common way of organizing P2P environments into distributed data stores in literature is by means of *Distributed Hash Tables* (DHT) [SMK+01, ZHS+04, RD01, RFH+01].

3.1 Distributed Hash Tables

The distributed hash tables are structured overlay networks that aim at integrating locally residing data stores at nodes of an execution environment into one distributed and decentralized system for the management of data. Thereby, the locally managed data stores of the participating nodes to the DHT have to be uniformly accessible to all nodes for lookup and updates. Usually, the local data has to be structured in such a way so that it semantically correspond to Key-Value data stores. That is, Key-Value data stores, such as for instance hash tables, maintain sets of Key-Value pairs that associate unique keys to data items in memory.

In order to integrate the various Key-Value stores into one entity all nodes that host them have to be subjected to a common, abstract address space, i.e., a common key identifier space KI for all data items. Thereby, the key identifier space KI is usually chosen to conform to a discrete metric space, such as for example a subset of the integer space (e.g., $KI = \{0..2^{32}\} \subseteq I\!N^+$) so as to address data items. Mapping of data items and nodes into the key identifier space is achieved by double hashing. Precisely, data items are mapped into KI by applying a consistent hashing function f_h onto the data item key value K. Those data items need to be assigned to a set of nodes N, hence all nodes of N have to be mapped into the same key identifier space KI, as well. Node mapping is achieved by applying another consistent hash function f_n onto the node identifiers, such as for instance node IP addresses. The following equation sums up the mandatory key identifier space mapping functions:

$$\exists f_h, f_n : f_h : K \mapsto KI \wedge f_n : N \mapsto KI$$

By mapping nodes and data item keys into the same key identifier space KI an inherent association among them does not exist by default, i.e., it is not clear how the keys are distributed among the nodes with respect to KI. Although keys and nodes share the same space KI, their mappings by means of f_h and f_n will not create a direct association among them by for instance mapping them to same values in KI, i.e., $f_h(k) \neq f_n(n)$. It can however occasionally occur that keys and nodes feature the same KI values (i.e., $f_h(k) = f_n(n)$), but this is not a general rule of thumb. Rather keys will stand alone in KI with respect to nodes. To associate keys to nodes directly, with respect to KI, even though no node is likely to be mapped to the same KI value to which a key is mapped, a special association function has to be introduced The function *assign* associates each data item key to only one node in the environment. Thereby, the actual assignment

logic is bound to the concrete implementation of the distributed hash table. The formal definition of the *assign* function is provided as follows:

Definition 3.1 (Key node allocation function – assign)**.** *Let K be the set of data item keys and N the set of nodes, then the allocation of keys to nodes is defined by the function assign with:*

$$assign : K \mapsto N$$

□

By applying the *assign* function on the data item keys set K each node will be assigned to a subset of K. Since each key possesses a mapping into key identifier space each node will be assigned with a subset of KI, as well. Given that *assign* maps keys to one node only, KI will actually be divided among the nodes into partitions. A formal definition of a key identifier space partition, assigned to a node, is provided as follows:

Definition 3.2 (Key identifier space partition – KIP)**.** *The key identifier space partition $KIP(n)$ of node $n \in N$ is defined as a subset of the key identifier space KI whose mapping data item keys are assigned to n:*

$$KIP(n) \subseteq \{KI | \forall ki \in KIP(n)\ \exists k \in K : f_h(k) = ki \wedge assign(k) = n\}$$

□

Therefore, the union of all key identifier space partitions along all nodes will result in the key identifier space itself and no two key identifier space partitions can have the same key identifiers encompassed:

$$\bigcup_{n \in N} KIP(n) = KI$$

$$\forall n, n' \in N : f_n(n) \neq f_n(n') \implies KIP(n) \cap KIP(n') = \emptyset$$

Given a key identifier value, finding the node that is responsible for it, such that it lies within the key identifier partition of this node, is facilitated by means of the function `resp`. Formally, `resp` is defined as follows:

Definition 3.3 (Key identifier node responsibility function – resp)**.** *The responsibility of keys identifiers of KI among nodes of N is defined by the function resp with:*

$$resp : KI \mapsto N \ \textbf{\textit{where}}\ resp(ki) = n \ \textbf{\textit{with}}\ ki \in KIP(n)$$

□

Only a node n responsible for a partition $KIP_{(n)}$ can offer data management operations (e.g., insert, update and delete) for data items mapping into it. Data items operations which are mapped onto key identifiers that lie outside of scope of its partition have to be delegated to the corresponding nodes. To connect the DHT nodes for efficient data management delegation purposes they have to be structured into a topological overlay network. Thereby, the applied topologies of the structured overlay networks can vary in efficiency [SMK+01, ZHS+04, RD01, RFH+01], in terms of data operation delegation, and are dependent on the actual application scenario of the DHT.

3.1.1 Ring Topologies

The considered structured overlay network for this thesis corresponds to a ring topology. The reason behind this lies in the fact that ring topologies are simple to construct and easy to maintain. For a ring topology to be constructed a DHT node n has to connect with only two adjacent nodes, a *predecessor* and a *successor* in the ring topology.

To determine adjacency among nodes of ring topology the key identifier space can be exploited. Namely, the *successor* succeeds node n if it is the first node that comes after n in KI space when going in a clockwise direction, starting from n. By going from the node with the highest mapped value in KI we would reach the end of KI in a liner space and thus remain without a successor. Hence, the liner KI has to be bent into a circular metric space by means of the *mod* operator (i.e., $mod|KI|$) such that any clockwise direction progression is restarted to the beginning of it when the end is encountered. This way the last node in the linear KI will be succeeded by the first node in the linear KI and thus the ring topology can be completed. In order to determine adjacency among the nodes inside the circular key identifier the node successor function *succ* is introduced. The function *succ* is formally defined as follows:

Definition 3.4 (Node successor function – succ)**.** *Let KI be the key identifier space. The adjacency within KI is defined by the function succ such that for any given node $n \in N$ it returns the first succeeding node in KI, i.e., the node $n' \in N$. Thereby, n' is the fist node encountered in the circular identifier space going in a clock-wise direction, i.e., starting from n:*

$$succ : N \mapsto N \text{ and}$$

$$succ(n) = n' \textbf{ with } n' = \arg\min_{x \in N}\{(f_n(x) - f_n(n)) mod |KI|\}$$

□

Given the adjacency of nodes, in terms of KI mapping, assignment of keys to nodes can be easily implemented in ring topologies by assigning all key identifiers that lie between two nodes to one of them. Usually all keys are assigned to the adjacent node with the higher key identifier space value. Hence, in ring topology distributed hash tables, the node responsibility function *resp* is implemented by associating keys identifiers with nodes in KI space that are encountered first by going in a clockwise direction:

$$resp(ki) = n \textbf{ with } n = \arg\min_{x \in N}\{(f_n(x) - ki) mod |KI|\}$$

The same applies for the assignment function *resp*. Keys can only be assigned to a node if their mappings into KI encounter that node first when going in a clockwise direction:

$$assign(k) = resp(f_n(k))$$

A key identifier partition thus features a monotonically increasing set of key identifiers that is characterized by a lowest and a highest value that bound it. Naturally, the lowest value corresponds to the first key identifier that comes after the predecessor node, whereas the highest value corresponds to the responsible node key identifier. Hence, a key identifier can thus be denoted by means of the two extremes as an interval, thereby implying the key identifiers that lie between them.

Corollary 3.5 (Key identifier space partition interval)**.** *Given a node $n \in N$, its responsible key identifier space partition $KIP(n)$, and a node $n_p \in N$ for which node n is the successor. Then $KIP(n)$ is denoted by means of the half closed interval (x,y] such that x is the lower bound and it corresponds to a key identifier space mapping of n_p, whereas y is the upper bound and corresponds to a key identifier space mapping of n:*

$$KIP(n) = (x, y] \textit{ where } f_n(n) = y \wedge f_n(n_p) = x \wedge succ(n_p) = n$$

□

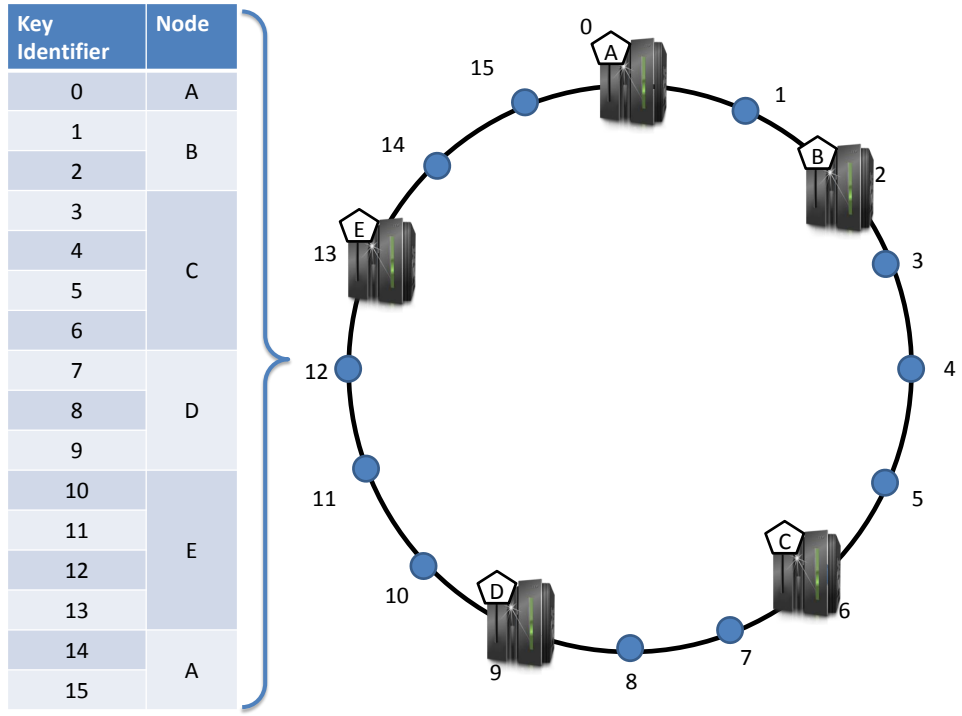
The simplicity in the ring topology maintenance thus lies in the fact that each node only has to actively maintain two links, namely the ones towards its predecessor and its successor node. The complex maintenance of the rest of the ring remains hidden from it behind its predecessor and successor nodes. In fact, ring topology maintenance is also very simple when new nodes are joining or existing nodes are leaving the overlay network. In case a node n leaves, then it just has to instruct its successor node to re-link with its predecessor and the topology is preserved without affecting the other nodes. On the other hand, if a node n wants to join, it merely has to interject between two corresponding nodes of a precedence relation by updating their predecessor-successor links towards it. The updated precedence order relations of the joining/leaving node n are determined by the function $succ^*$ such that it is a transitive closure of $succ$.

When it comes to data management, the joining and leaving of nodes in ring topologies results in the splitting and merging of the corresponding data partitions. Naturally, data partition merging and joining is accompanied with the transfer of the encompassing data items among the nodes.

Example 3.1

Figure 3.1 illustrates an example of a ring topology overlay network. As the figure shows there are five nodes A, B, C, D and E such that node n_A maps to key identifier space value 0, n_B to 2, n_C to 6, n_D to 9 and n_E to 13. The assumed key identifier space is of size 16, i.e., $KI = \{0..2^4\}$. The corresponding precedence order is thus as follows: $n_B = succ(n_A), n_C = succ(n_B), n_D = succ(n_C), n_E = succ(n_D)$ and $n_A = succ(n_E)$. Given this mapping of nodes into KI, the data item partitions for which the node are responsible are resulting as follows: $KIP_A = (14, 0]$, $KIP_B = (1, 2]$, $KIP_C = (3, 6]$, $KIP_D = (7, 9]$ and $KIP_E = (10, 13]$. The table displayed at left hand side of the figure illustrates KI in the first column and its assignment to nodes in partitions in the second column. In case of a node leave, for instance of node C, than node B would precede node D instead and the key identifier space partition KIP_C would merge with KIP_D, i.e., $N \setminus C \Rightarrow n_D = succ(n_B), kip_D = (3, 9]$.

The simplicity of the ring topology, in terms of construction and maintenance, however does not come without a price. In case a node is accessed with an operation request (e.g., insert, update or delete) for a data item key which lies outside of its key identifier space partition than this request has to be delegated to another node. Since each node is not aware of the whole ring but rather only of its adjacent nodes in the topology it



Key Identifier	Node
0	A
1	B
2	
3	C
4	
5	
6	
7	D
8	
9	
10	E
11	
12	
13	
14	A
15	

Figure 3.1: Key identifier partitions of the ring topology

can only pass on the request to either one of them. This routine has to repeated at all nodes along the chain of links until the corresponding key identifier space and its owning node have been found. Thereby, the delegation has to be preformed in a directed manner, i.e., if data access requests that are coming from the predecessor can only be passed on to the successor and vice-versa. The initial direction of the delegation propagation is usually based on a greedy decision. Precisely, the access request is passed on to the adjacent node whose key is closer to the data access key inside the key identifier space. Random selections of the direction are also possible.

Although this data access routine is fairly simple to realize it does not always offer optimal DHT performance. For instance, the greedily selected direction can in the end turn out to feature more nodes along the path than a randomly selected path. Moreover, if frequently accessed data is located at distant nodes, which have to be reached over numerous node traversals along the path, then the DHT is going to feature a bad data access performance as well. In general, the asymptotic complexity of this data access routine is linear with the number of nodes in the DHT environment, i.e., $O(N)$. When applied to huge environments that are composed of a very big number of nodes the greedy data access algorithm yields inadequate performance to a level which makes it useless. This is why distributed hash tables have to be enhanced with more efficient data access algorithms that are insusceptible to big node numbers in the environment. Precisely, we consider the well researched Chord [SMK+01] data access protocol for distributed hash tables.

3.1.2 Chord

In order to improve the performance of a DHT the number of traversed nodes has to be minimized for every data access request. Thereby, the number of the traversed nodes directly depends on the key identifier space distance of the requested key to the node at which the request was issued. Assuming a uniform distribution of key identifier space partitions, in terms of size, among the nodes a greater distance towards the requested key will always result in more nodes for traversal in delegating the access request. Note, that by applying the consistent hashing function the uniform partition assumption is always inherent as such hashing functions guarantee even distribution of key identifiers among the number of partitions, i.e., nodes that are responsible for them. While shorter distances should feature lesser nodes for traversal and acceptable performance, greater distances should generally feature more nodes and thus unpredictable performance.

To overcome this problem, assuming that continuous data redistribution to close nodes is not feasible, shortcuts have to be introduced that can advance the data access requests faster to the destination nodes. In view of this, DHT nodes have to additionally maintain a list of shortcuts, next to the predecessor and successor nodes, which will bring them faster to distant key identifier partitions. Maintaining the shortcuts list however for all distant key identifier space partitions is not easy for execution environments of numerous nodes. This is why the shortcut lists have to feature only a limited number fast paths that allow for coverage of big distances inside the key identifier space while traversing the nodes for the destination key.

The main idea behind Chord is to maintain a list of forwarding nodes, also referred to as the *Finger Table*, that will feature for any key identifier value a reduced distance to it of at least by a half. A node belongs to the Finger Table from the perspective of node n only if it is the first node in KI that reduces the distance by a factor of 2 for some given key identifier. By the same logic, if we add a distance of the factor 2 to node n than the first node that comes after this distance is a Finger Table node. A formal definition of the Finger Table node is provided as follows:

Definition 3.6 (Finger Table node – fn)**.** *Let $i \in I\!N^+ \cup \{0\}$ be the distance defined by 2^i which is added in KI to a Chord node $n \in N$. A Finger Table node of n is the first node that comes after the added distance of 2^i. The Finger Table node $fn(i)$ of n, given the distance 2^i is defined as follows:*

$$fn(i) = resp((f_n(n) + 2^i) mod |KI|)$$

□

The biggest possible distance of a key identifier space is the size of the key identifier space itself. If we select this size to be an exponential number of the base 2 (i.e., $|KI| = 2^m$, where $m \in I\!N^+$) then the coverage of all of KI by iteratively halving it takes at most $log(2^m)$ iterations. This implies that only m (i.e., $log(2^m) = m$) Finger Table nodes are needed so as to address the whole key identifier space of size 2^m. The Finger Table is formally defined as follows:

Definition 3.7 (Finger Table – FT)**.** *Let the key identifier space be of size* 2^m *where* $m \in IN^+$ *and* $KI = [0..2^{m-1}]$*. The Finger Table* $FT(n)$ *of node* $n \in N$ *is defined as a set of m finger nodes. Each finger node is obtained by adding m distances of* 2^i *one at the time to node n:*

$$FT(n) = \bigcup_{i=0}^{m-1} fn(i)$$

The set of all node Finger Tables is defined as FT. □

The Chord protocol thus constructs the Finger Table by partitioning the key identifier space into m intervals of exponentially increasing size. Each Finger Table interval is defined by a start and an end. The start of an interval is obtained by adding a distance of the factor 2 the node. The end of an interval is obtained as the start of the succeeding interval, whereas the node that is responsible of this interval is the corresponding fn. Table 3.1 summarizes the structure of a Finger Table entry as follows:

Table 3.1: Finger Table $FT(n)$ entries for node n, part of a key identifier space of 2^m.

Notation	Definition
$fn(i).start$	$(f_n(n) + 2^i)\ mod\ 2^m, 0 \leq i \leq m-1$
$fn(i).end$	$fn(i+1).start$
$fn(i)$	$resp(fn(i).start)$
successor	$fn(1)$
predecessor	predecessor node of the ring topology, i.e., n_p

Given a key identifier lookup at a node, first the Finger Table is consulted for the interval that contains the key identifier. In case the encompassing interval is assigned to the successor of the lookup node, the search is concluded in declaring the successor responsible for the data item. Otherwise the same Finger Table lookup is performed at the node responsible for the encompassing interval until the corresponding successor is reached. Since the Finger Table intervals are constructed to be exponentially increasing in size the encompassing interval node will feature a distance to the corresponding successor that is reduced by at least a half. By repeating the Finger Table lookups and thus reducing iteratively the distance to the corresponding successor by a half the search will be concluded eventually in a number of steps that is logarithmic in the size of the key identifier space, i.e., $log(2^m)$. Given a huge node environment, data lookup routines of a logarithmic asymptotic complexity (i.e., $log(2^m)$) can be considered as extremely efficient.

Algorithm 3.1 and Algorithm 3.2 provide the corresponding pseudo-code that conducts Chord based data access request forwarding. Note, that Algorithm 3.1 corresponds to an implementation of the function $resp$.

Example 3.2

An example of a Chord ring topology overlay network is illustrated in Figure 3.2. As

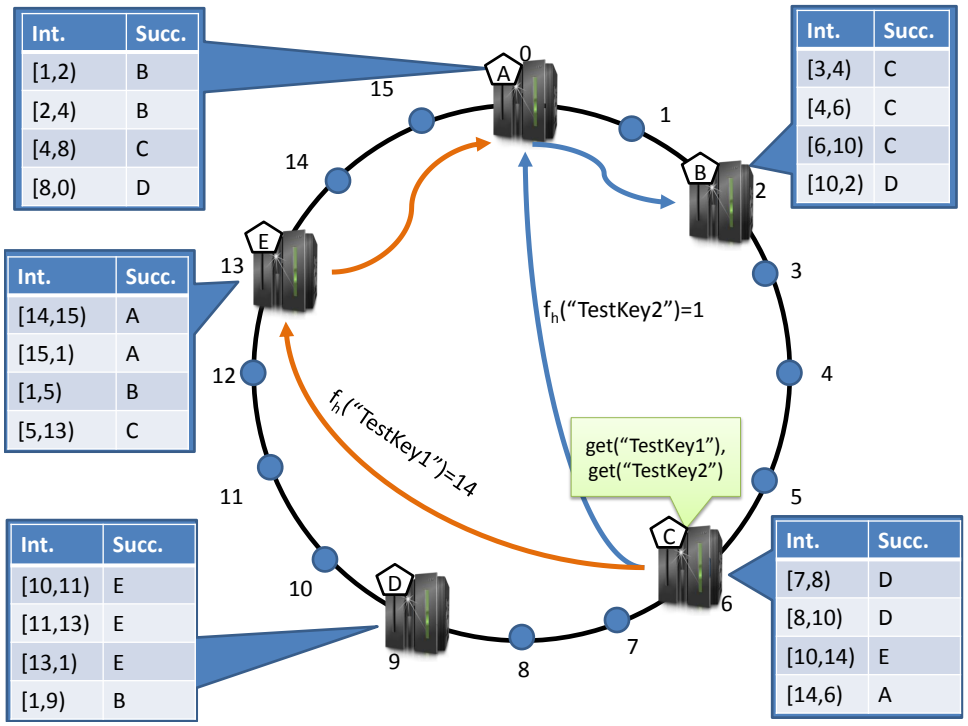


Figure 3.2: Chord data access forwarding

Algorithm 3.1 *n.findSuccessor(id)* returns at node n the successor for the key identifier id

```
1: if id ∈ (n, successor] then
2:     return successor
3: else
4:     n0 ← closestPrecedingNode(id)
5:     return n0.findSuccessor(id)
6: end if
```

we can see this figure shows the Finger Tables for each node in the ring. Moreover, it builds upon the semantics of the ring topology, in terms of node mappings and partition allocations in KI, presented in Figure 3.1. Thereby, each Finger Table features the key identifier space intervals and their first successors. The applied key identifier space is of size 2^4, hence the Finger Tables are of size 4 (i.e., $m = 4$). Of course, the starting values of the intervals are computed by means of the formula of Table 3.1. For instance, for node A, which maps to the key identifier 0, the starting intervals of the Finger Table are computed as follows: $fn(1).start = (0 + 2^{1-1})\ mod 16 = 1$, $fn(2).start = (0 + 2^{2-1})\ mod 16 = 2$, $fn(3).start = (0 + 2^{3-1})\ mod 16 = 4$ and $fn(4).start = (0 + 2^{4-1})\ mod 16 = 8$. Regarding the successors, the figure shows that the first successor for $fn(1) = B$ (maps to key identifier 2), $fn(2) = B$ (maps to key identifier 2), $fn(3) = C$ (maps to key identifier 6) and $fn(4) = D$ (maps to key identifier 9). The remaining Finger Tables are computed in the same way, they are just offset by their key identifier value. As the figure shows, if for instance node C is confronted with a data access request of the key value "TestKey1" it would first have map it into the key

identifier space by means of the hash function obtaining thus the key identifier value 1. By looking up in its Finger Table, by means of the `findSuccessor()` algorithm, it would find out that the key identifier 1 is located within the interval $[14, 6)$ which is assigned to node A. As a next step it would forward the data access request to node A which would also perform the same routine in executing `findSuccessor()`. However, at node A the Finger Table lookup would return the successor as the node whose interval encompasses the key identifier 1 in which case the data access delegation is concluded with the node B being the responsible one.

Algorithm 3.2 *n.closestPrecedingNode(id)* returns first successor of *id* from the Finger Table of node n.

```
1: for i ← m downto 1 do
2:     if id ∈ [n.finger_i.start, n.finger_i.end) then
3:         return n.finger_i
4:     end if
5: end for
6: return n
```

In order to keep the Finger Table updated with optimal nodes, Chord has to continuously run a set of routines that periodically validate the current Finger Table nodes for their position in the ring topology with respect to node joins/leaves in the environment. Naturally, each Chord node has to continuously validate its predecessor and successor nodes so as to maintain the correct structure of the ring topology, in terms of the precedence order of the key identifier space. In doing so, each DHT node has to continuously check whether its predecessor node is alive, by means of periodical heart beat messages. In case it is not the precedence order has to be dissolved by removing the unresponsive predecessor from the Finger Table. Algorithm 3.3 shows the predecessor validity check routine.

Algorithm 3.3 *n.checkPredecessor()* periodically updates the predecessor at node n with respect to its liveliness.*id*

```
1: if predecessor has failed then
2:     predecessor ← null
3: end if
```

Moreover, a DHT node has to continuously check its successor node for the precedence order consistency among them. This means that a node checks its successor by periodically querying it for the node that precedes it. In case there is consistency conflict in the precedence order among a node and its successor, the successor will show to a different predecessor node than the node that issued the query. A node can repair its successor node precedence order conflict by advertising at the new predecessor node of the successor with itself as its predecessor. Consistency of the precedence order is restored and so is the correct ring topology from a Finger Table perspective, in case the advertising node is accepted as the predecessor at the successor's predecessor.

Algorithm 3.4, also referred to as `stabilize()`, shows the precedence order consistency check routine, whereas Algorithm 3.5, also referred to as `notify()`, shows the precedence order repair routine.

Algorithm 3.4 *n.stabilize()* periodically checks the predecessor of the successor at a node n and in case it is not n updates the successor of n.

1: $x \leftarrow successor.predecessor$
2: **if** $x \in (n, successor)$ **then**
3: $\quad successor \leftarrow x$
4: **end if**
5: $successor.notify(n)$

Algorithm 3.5 *n.notify(n′)* checks at node n whether a node n' could be its predecessor and if so then it updates the predecessor to n'.

1: **if** $predecessor = null \vee n' \in (predecessor, n)$ **then**
2: $\quad predecessor \leftarrow n'$
3: **end if**

The subsequent changes to the ring topology, in terms in the updated predecessor and successor node precedence orders, are propagated to the rest of the ring only reactively. This means that the Finger Table updates are not directly sent to all other nodes of interest but rather that the updates have to be retrieved by the nodes themselves. In Chord this is achieved by means by periodical lookups for nodes that are responsible for specific key identifier values. Recap, a Chord data access lookup, by means of iterative distance halving, terminates at a node that concludes its successor to be the destination node. Hence, updates to the ring topology that are changed by means of the `stabilize()` will be reflected with different successor nodes for the same lookup key identifier. To keep the Finger Table updated, Chord thus has to periodically lookup for the successor of its Finger Table interval start key identifier values. Algorithm 3.6, also referred to as `fixFingers()`, shows the interval start successor lookup routine.

Algorithm 3.6 *n.fixFingers()* periodically searches for the successors of the Finger Table interval start values at node n.

1: $i \leftarrow i + 1$
2: **if** $i > m$ **then**
3: $\quad i \leftarrow 1$
4: **end if**
5: $finger[i] \leftarrow findSuccessor(finger[i].start)$

In order for a node to join an existing Chord ring a bootstrapping node that is already part of the ring structure is necessary. The bootstrapping node can refer the joining node to its possible successor, by means of a `findSuccessor()` lookup, so that the joining node can start the `stabilize()` routine. Algorithm 3.7 shows the Chord ring node joining routine.

Algorithm 3.7 *n.join(n_b)* queries the bootstrapping node n_b for an successor at node n

1: $predecessor \leftarrow null$
2: $successor \leftarrow n_b.findSuccessor(n)$

The introduced algorithms of Chord provide a simplistic yet powerful extension to the ring topology overlay networks that guarantees scalable data access performance. However, the Chord extension does not fully outfit ring overlay networks for all behavioral aspects of a distributed peer-to-peer system. For instance, at times a the ring overlay networks might be subject to very high frequencies of concurrent node joins and leaves. The price of a rather simplistic approach in such scenarios reflects in inconsistencies at Finger Tables across a big set of nodes. This drawback of Chord is known as the *Churn* problem and has been extensibility addressed in literature [KEAAH05, RGRK04]. Despite the churn drawback, theoretical analysis of Chord by the authors proves that inconsistent Finger Tables eventually converge from even very large numbers of node joint/leaves providing in the process mostly correct lookups and with a high probability of $O(log 2^m)$ node forwarding steps. Another notable drawback of Chord is its rigid application of the same hash function for both data and node mapping into the key identifier space. Although such an approach is undeniably straightforward it disregards the physical characteristics (e.g., locality) of the hashed nodes. For instance, nodes that are physically proximate might be mapped across great distances within the key identifier space, causing this way additional node traversals among them which would normally not be necessary. To overcome this issue, the application of local sensitivity hashing functions [HMA09] has been considered in the past. Hence, for all of Chord's benefits, in terms of simplicity and scalable lookup performance, a significant body of research seeking to overcome all of its drawbacks has been sparked thus ensuring a secure future for Chord both in academia as well as in industry.

3.2 DHT Data Availability

So far we have discussed data inside the peer-to-peer networks in the context of efficient addressing and lookup but have disregarded actual physical location of data. To convey the data that is subject to management by a peer-to-peer system data items are introduced. A data item is formally defined as follows:

Definition 3.8 (Data item – di)**.** *Let Φ be the common data alphabet used for the encoding of payload information. A data items di is defined as tuple $di = (k, d)$ where $k \in K$ is the data item key and $d \in \Phi$ is the payload information encoded with the common data alphabet. The set of all data items is denoted as DI.* □

Structured peer-to-peer networks imply data items be allocated only within the encompassing key identifier space partition and thus the responsible node. A distributed hash table can thus be formally defined as follows:

Definition 3.9 (Distributed hash table – DHT)**.** *Given the the set of all nodes N, the set of data items DI and the set of key identifiers KI, the distributed hash table DHT is defined by the*

relation $DHT \subseteq (DI \times N \times KI)$ *such that for each data item there exists one responsible node that stores it:*

$$\forall di \in DI \; \exists n \in N : (di, n, ki) \in DHT \wedge assign(di.k) = n \wedge f_h(di.k) = ki \wedge resp(ki) = n$$

□

However, given an execution environment of high dynamics, the assumption of storing one data item per node has to be loosened. In particular, if a node environment is of dynamic character and features high node join/leave frequencies due to the physical failures, certain measures have to be taken to assure the physical availability of data. Distribution of all data items across a set of nodes by means of replication is the most common way to facilitate data availability in the presence of node failures. Naturally, the more replicas of a data item are created and disseminated in the node environment the more node failures the system can sustain without loosing the data completely. However, more replicas also incur more overhead on the system by physically allocating replicas to nodes.

The data availability semantics of the system are subject to a replication factor r that determines the number of replicas for each data item to be allocated at different nodes. Thereby, the replication factor can range from just one replica to as many replicas as the size of the key identifier space. Each replica is distinguished by an identifier which reflects an enumerator of the replication factor r. The set of replication factor enumerators is summarized as follows:

$$R = \{1, ..., r\} \text{ with } 1 \leq r \leq |KI|$$

Depending of the specific DHT implementation, there are many ways to allocate replicas to DHT nodes, all of which bear certain advantages over the others when overhead is considered. Precisely, the most simplistic approaches to data availability in a Chord DHT are the *multiple hash functions* replication and the *successor-list* replication schemes.

3.2.1 Multiple Hash Functions

As the name immediately suggests multiple hash functions replication is based on exploitation of a set of distinct hash functions H so as to allocate the same data item into the same key space. Thereby, the cardinality of the leveraged hash functions set corresponds to the replication factor r the DHT wants to achieve. The hash function set is summarized as follows:

$$H(r) = \{f_{h1}, f_{h2}, ...f_{hr}\}$$

A data item that is supposed to be redundantly stored at different nodes is hashed by means of all hash functions so as to obtain different key identifier space values. The responsible nodes of the different key identifier space values can be found by means of Chord and subsequently used for redundant data storage. To physically store a data item *di* at a node *n* the routine `n.put(di)` is used. Algorithm 3.8 shows the multiple hash function routine.

Algorithm 3.8 *n.multipleHashes(di)* applies all hash functions on the given data item *di* at node *n* and subsequently determines the replication node upon which the local data allocation operation is used.

1: **for** $f_{hi} \in H$ **do**
2: $ki_{hi} \leftarrow f_{hi}(di.k)$
3: $succ_{hi} = n.findSuccessor(ki_{hi})$
4: $succ_{hi}.put(di)$
5: **end for**

Multiple hash functions offer a low overhead, since only the normal way to data storage has to be applied multiple times. However, they also feature a set of drawbacks. In practice, it is very difficult to devise good hash functions that will always map keys to different nodes. Occasionally two different hash functions can map data items to key identifier space values that are within responsibility of the one node:

$$resp(f_{h1}(k)) = resp(f_{h2}(k))$$

As a consequence, the redundancy of the data items is lesser than the desired replication factor r, thus the system is less robust to node failure. Moreover, the replication factor of a data item cannot be maintained in the event of a node failure. If a node holding a replica fails, then its replica is also lost and cannot be restored at the successor since the other replicas are disseminated at random nodes with no relation to the failed node whatsoever. Only an update to the compromised data item by the end-user can restore the replication factor again.

3.2.2 Successor Lists

For the aforementioned reasons of the previous section, successor-list replication schemes are generally preferred to the multiple hash functions in Chord. Still other DHT systems such as CAN [RFH+01] and Tapestry [ZHS+04] are relying on multiple hash functions for replication purposes. On the other hand, in the successor-list approach a node's Finger Table is extended with a set of subsequently succeeding nodes in the key identifier space. The nodes of the successor-list are exploited for replication. Thereby, the size of the successor-list corresponds to the replication factor r. Hence, the successor list is summarized as follows:

$$SL(r) = \{n_1, ...n_r\} \text{ with } n_1 = succ(n) \wedge n_2 = succ(n_1) \wedge ... \wedge n_r = succ(n_{r-1})$$

$$(di, n_1, ki) \in DHT \implies \forall n_i \in SL \textbf{ with } 2 \leq i \leq r \ (di, n_i, ki) \in DHT$$

Successor-lists are the standard approach to replication in Chord although they violate the assignment function `assign()` by allocating data items to nodes that should not be there. This violation however is beneficial in the event of a node failure as it features seamless key space partition responsibility hand over to the successor. Precisely, in case a node fails, then its first successor in the list already contains its replicated data items. The first successor can thus automatically take over responsibility of the failed

predecessor's key identifier space partition by repairing the ring topology precedence order in updating the predecessor. In contrast to multiple hash tables, successor-lists feature replication factor restoration of a data item which improves its robustness to node failure. Precisely, a failed node that is a member of a predecessor successor-list is replaced by extending the affected successor-list with one additional node and removing the failed node. Algorithm 3.9 shows the successor-list update routine.

Algorithm 3.9 *n.updateSuccessorList(n_{fail}, SL, di)* removes the failed node n_{fail} and adds a new member to the successor-list SL such that it succeeds the last successor-list member. Afterwards a given data item *di* is added to it.

1: **if** $n_{fail} \in SL$ **then**
2: $SL \setminus n_{fail}$
3: $n_{last} \leftarrow SL[r]$
4: $n_{new} \leftarrow n.findSuccessor(f_n(n_{last}) + 1)$
5: $SL \cup n_{new}$
6: $n_{new}.put(di)$
7: **end if**

Although the replication factor restoration of a data item seems fairly simple it is however afflicted with a significant overhead. In the event of a failure, all preceding nodes that contain the failed node have to repair their successor-lists by means of the `updateSuccessorList()` routine. This implies, that r preceding nodes are directly always affected by the failure of a node, where f is the replication factor. Moreover, not all nodes will immediately be able to restore the successor-lists correctly. It takes some time till successor-lists of the predecessors converge. The predecessor of the failed node has to repair the precedence order of the ring topology and propagate this update to its predecessor and so on. Usually, r `stabilize()` routines invocations are necessary by each of the r failed node predecessors till the successor-lists converge, i.e., $O(r^2)$ stabilization messages in total. During this time the system is susceptible for additional failures which could bring the system into an irreparable state.

3.2.3 Symmetric Replication

To overcome replication factor restoration overhead issues, additional approaches to data availability in Chord have been extensively studied in literature. One of them features the *symmetric* replication [GAH07] of data items across the circular key identifier space of Chord. The idea behind symmetric replication is to affect the smallest possible amount of nodes while recovering a failure and thus keep the replication factor restoration overhead low. To do so, symmetric replication bases its replication strategy along the key space identifiers and not the nodes themselves.

The key identifier space *KI* is always fixed and the key identifiers will unlike nodes never join or leave. Only the nodes that are responsible for them are subject to change. However, as only one node is responsible for any key identifier, a change in responsibility only affects one node (i.e., the newly responsible one) and not more than that. Hence, the overhead, in terms of affected nodes, is by itself lower.

To facilitate replication based on key identifiers, each one of them is associated with $r-1$ other replica key identifiers. The set of r associated key identifiers constitutes an equivalence class. Hence, each equivalence class features a set of key identifiers that are bound to each other. Distinction among the equivalence class key identifiers, is facilitated by means of an association to a replica enumerator. Equivalence classes thus groups key identifiers that are each associated to a distinct replica enumerator. To group key identifiers into equivalence classes a mapping function is needed that guarantees association among all key identifiers which is uniform, unique and disjoint. The equivalence class is formally defined as follows:

Definition 3.10 (Symmetric replication equivalence class – *ec*)**.** *An equivalence class ec is defined as tuple* $ec = (KI_{ec}, R, symm)$ *where:*

- $KI_{ec} \subseteq KI$ *is the set of equivalence class key identifiers,*
- R *is the replication factor enumerator set.*
- *symm is the mapping function that associates a key identifier* $ki \in KI$ *based on a replica enumerator* $i \in R$ *to a congruent key identifier* $ki_{ec} \in KI_{ec}$ *of the same equivalence class:*

$$symm : KI \times R \mapsto KI$$

$$\forall ki_{ec} \in KI_{ec}\ \exists i \in R, ki_{ec} \in KI_{ec} \implies symm(ki, i) = ki_{ec}$$

The set of all unique equivalence classes is defined as EC. □

Uniqueness of equivalence classes in symmetric replication is achieved by implementing the function *symm* as a congruence class modulo r operator as follows:

$$symm(ki, i) = ki_{ec} \textbf{ with } ki_{ec} = (ki + (i-1)\frac{|KI|}{|R|}) mod |KI|$$

Hence, a key identifier *ki* is said to be *congruent* with ki_{ec} for the i-th replica with respect to the same equivalence class.

All data items of an equivalence class, in terms of the same equivalence class key identifier ki_{ec}, have to be collocated at the same node at all times. Moreover, each equivalence class has to be shared and redundantly maintained by r nodes such that each redundant node corresponds to a responsible of some data item of the equivalence class. A DHT which adheres to the symmetric replication distribution of data items is formally defined as follows:

Definition 3.11 (Symmetric replication DHT – DHT_s)**.** *Given the distributed hash table set DHT, the equivalence class set EC and the set of all node Finger Tables FT. A symmetric replication distributed hash table is defined as the relation* $DHT_s \subseteq DHT \times EC \times FT$ *where*

$$\forall (di, n, ki) \in DHT\ \exists ec \in EC : \forall i \in ec.R, symm(f_h(di.k), i) \in ec.KI_{ec}$$

□

Regarding data availability in face of node failures, traditional replication systems necessitate a higher replication factor so as to guarantee a better robustness. Naturally, the more replicas there are in the system the more failures it can sustain. The only downside to a high replication factor is the consumed time that is needed to allocate all replicas at all nodes. In the case of Chord, all responsible nodes have to be found first by means of the logarithmic lookup routine `findSuccessor()` before they can be copied. What makes symmetric replication really stand out from other traditional approaches is the fact that data availability can be restored more efficiently than with the others in the event of a failure. Symmetric replication is more robust to node failures, as replication factor restoration on average affects only one other node.

Whenever a node leaves the DHT environment (by itself or due to failure) each of its assigned key identifiers can be retrieved from another congruent key identifier. That is, the successor of a departed node can compute for each of newly assigned key identifier one congruent key identifier and query its responsible node (via Chord) for the whole equivalence class and all of its encompassed data items. Once the equivalence class, and all its data items, have been received the replication factor is restored.

Not only is it efficient to restore one key identifier, but also the whole key identifier partition can be efficiently restored by affecting only one other node. Given an uniform size distribution of key identifier partitions among nodes (as provided by the SHA-1 hash function) the key identifier space partition of the departing node should be symmetrically mirrored at some other node at the other end of the key identifier space as a set of congruent key identifiers. This implies that only one symmetric node needs to be contacted so as to obtain the whole key identifier space partition of the departed node. In other words, the expected overhead of the symmetric replication algorithm, in terms of exchanged messages, for the purpose of replication factor restoration is on average one, i.e., $O(1)$. The superior efficiency, as compared to the traditional approaches, enables the symmetric replication approach to apply a lower replication factor (e.g., $r = 3$–5) in order to guarantee availability of data.

The allocation of data items at nodes w.r.t. symmetric replication also violates the assignment function `assign`. Instead `assign` is replaced by `symm` function to as allocate data items at nodes. In using the *symm* operator nodes disseminate their locally stored data items as replicas inside the DHT environment by first determining the congruent equivalence class key identifiers and then by finding the their responsible nodes. To store data items at the equivalence class responsible nodes the routine $put_s(di, ki_{ec}, i)$ is used. The symmetric replication data item insertion routine is shown by means of Algorithm 3.10.

Algorithm 3.10 *n.putData(di)* applies the *symm* operator for all replicas of the replication factor set R on the given data item di at node n and subsequently determines the replication node upon which the the local data allocation operation is used.

1: **for** $i \in R$ **do**
2: $\quad ki_{ec} \leftarrow n.symm(f_h(di.k), i)$
3: $\quad succ_{ec} \leftarrow n.findSuccessor(ki_{ec})$
4: $\quad succ_{ec}.put_s(di, ki_{ec}, i)$
5: **end for**

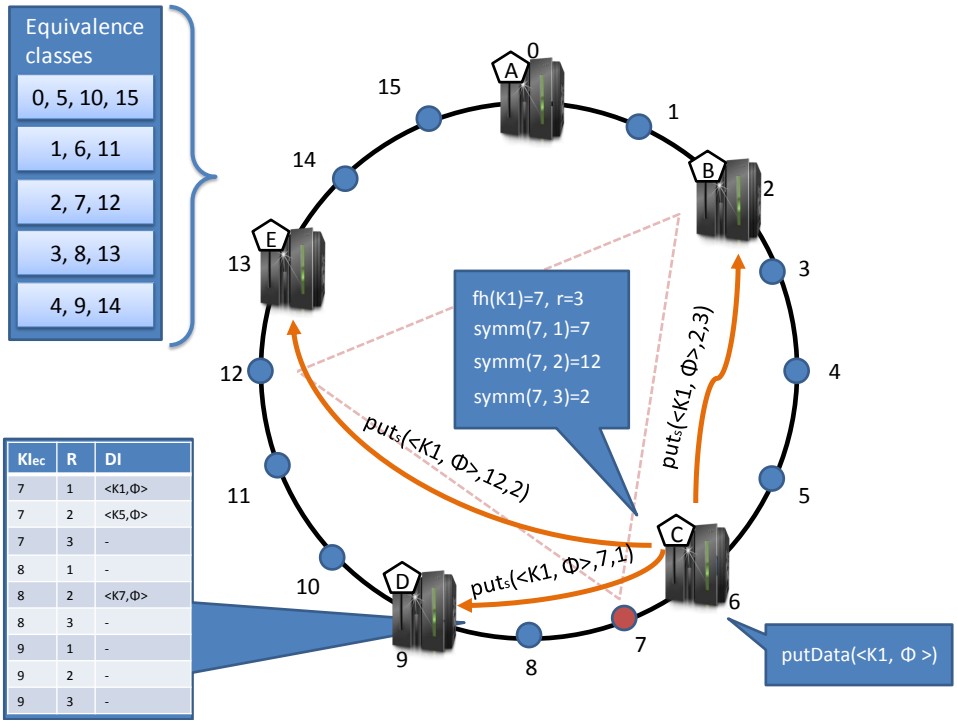


Figure 3.3: Symmetric replication `putData()` execution example.

Example 3.3

Figure 3.3 shows an example of an symmetric replication data insertion by means of Algorithm 3.10. This example builds on the same Chord DHT execution environment, i.e., the same environment configuration, as introduced in Figure 3.2. As illustrated in this example the key identifier space is of size 16 (i.e., $|KI| = 16$) and the replication factor is chosen to be 3 (i.e., $R = \{1,2,3\}$). As a consequence, five distinct equivalence classes (i.e., $\frac{|KI|}{r} = \frac{16}{3} \approx 5$) are created by means of the *symm* operator, which are displayed in the upper left box. Given the replication factor of three all equivalence classes contain also three key identifiers except for one. Due to the fact that the replication factor is an odd number and the key identifier space is an even number the key identifiers cannot be evenly distributed. Hence, one residue key identifier space has to be assigned to the first equivalence class. For instance, the equivalence class of the key identifier KI_7 is obtained by applying the *symm* operator r times for each member of the replication factor enumerator set R, i.e., $symm(7,1) = 7$, $symm(7,2) = 12$ and $symm(7,3) = 2$. The equivalence class congruence for KI_7 is depicted with the dashed triangle at which the triangle points correspond to the equivalence class key identifiers.

Moreover, this figure shows the locally maintained key-value store of node D with respect to the key identifiers and the replica iterator. For instance, under the key identifier 7 and replica iterator 1 a data item of key K_1 and payload Φ is stored. i.e., $D.get_s(7,1) = \langle K1, \Phi \rangle$. Whereas, the key identifier 7 with replica identifier 3 does not contain any data item associated to it, i.e., $D.get_s(7,3) = \emptyset$. Since the data item mapping key identifier 7 is congruent with the key identifiers 2 and 12 with respect to the

equivalence class, the responsible nodes of them have to share this data item as well. That is, node E, responsible for $ID = 12$, and node B, responsible for $ID = 2$, also share the data item $\langle K1, \Phi \rangle$ locally, albeit under different replica identifiers.

Algorithm 3.11 *n.updatePredecessor(n′)* applies *symm* on the interval with extended key identifier space partition, determined by the new predecessor n', and sends queries to the replica node for the congruent interval of the same equivalence class, by means of $obtainInterval()$.

```
1: end ← f_n(predecessor)
2: start ← f_n(n′)
3: predecessor ← n′
4: if start < end then
5:     x ← n.symm(start, 2)
6:     y ← n.symm(end, 2)
7:     n_rsp ← n.findSucc(x)
8:     n_rsp.obtainInterval(x, y, n)
9: end if
```

Whenever a node is to extend its key identifier space partition due to a change of predecessor, it can compute the equivalence class congruent bounds x and y, in terms of key identifiers, by means of the *symm* operator. The congruent bounds of the extended predecessor interval can be used to determine the responsible replication node n_{rsp}, by finding the successor of the upper bound x or the lower bound y. Afterwards, the responsible node n_{rsp} for the congruent bounds can be queried for all data items lying within the congruent bounds x and y. Algorithm 3.11 shows the update routine of the predecessor, and the consequent replication factor restoration.

Algorithm 3.12 *n.obtainInterval(x, y, n′)* given the bounds x and y selects all data items located within these bounds at node n and returns them to the specified node n' by means of the $replicate()$ operator.

```
1: DataItems[][] ← ∅
2: for i = x to y do
3:     for j =∈ R do
4:         di_cg ← n.get_s(i, j)
5:         DataItems[i][j] ← di
6:     end for
7: end for
8: n′.replicate(DataItems)
```

Once a replication node receives a query for an interval it selects all locally residing data items that are located within the specified bounds. Thereby, it selects all replicas that are associated to the key identifiers with respect some the equivalence class. To retrieve locally stored data items with respect to an equivalence class and a replication

factor the routine $get_s(ki_{ec}, i)$ is introduced. All encompassed data items can be sent back to the querying node with just one message, thus significantly reducing the replication factor restoration overhead. Algorithm 3.12 shows equivalence class data item selection routine.

Algorithm 3.13 *n.replicate(x, y, DataItems)* stores at node n, all received *DataItems* for the requested interval $x - y$ after rearranging them so as to match the equivalence class key identifier order and replica identifiers of the local node n.

```
1: for i = x to y do
2:     it ← n.symm(i, r)
3:     for j ∈ R do
4:         di_cg ← DataItems[i], [j]
5:         rep ← n.symm(j, r)
6:         n.put_s(di_cg, it, rep)
7:     end for
8: end for
```

Note, that in case the specified bounds of the interval to be obtained are not fully encompassed by the congruent node itself, the same `obtainInterval()` request can be further propagated along the successors node chain till the interval is fully matched, i.e., $y \leq f_n(n.successor)$. After the received data items have been rearranged so as to match the equivalence class order of the key identifiers, they are locally stored and the replication factor is successfully restored. Algorithm 3.13 shows the local replication factor restoration routine.

Algorithm 3.14 *n.joinReplication()* applies at obtainInterval() routine at the *successor* node of node n after having determined the interval bounds.

```
1: end ← f_n(predecessor)
2: start ← f_n(n)
3: successor.obtainInterval(end, start, n)
```

In case the predecessor of a node is updated to reduce the owning key identifier space partition in `updatePredecessor()` (i.e., Algorithm 3.11, line 4) this implies that a new node has joined the environment. In such a situation the predeccessor updating node only has to take part a passive role in the replication factor restoration. Rather, the responsibility over it is completely handed over to the newly joined node, which has to determine its key identifier space partition bounds and obtain the corresponding data items from its new successor with an `obtainInterval()` request. Algorithm 3.14 shows the node join routine.

Example 3.4

Figure 3.4 shows an example of a replication factor restoration process in face of a node leave. This example builds of the symmetric replication system already shown in Figure 3.3. Regarding runtime execution, Figure 3.4 depicts the departure of node

B due to failure, which triggers the node C to update its predecessor. Once the precedence order among nodes A and C has been repaired by means of the `stabilize()` routine, node C triggers the replication factor restoration, by means of the routine `updatePredecessor()` as follows:

1. First node C determines its key identifier space partition $KIP_c$to be extended by the key identifier partition of node B bounded by the interval $[1,2]$ of, i.e., $N \setminus B \Rightarrow n_c = succ(n_a), KIP_c = (0,9]$.

2. Afterwards, node C determines the interval $[1,2]$ to be congruent with the interval $[6,7]$ with respect to the equivalence classes by means of the *symm* operator. Since C is responsible for the congruent interval lower bound (i.e., $x = 6$) itself it is the first responsible node for querying by means of the `obtainInterval()` routine. However, since C does not encompass the whole congruent interval it forwards the `obtainInterval()` request to its successor, i.e., node D.

3. Node D encompasses the queried interval end bound (i.e., $y = 7$) and returns all data items located within the interval with one message to node C by means of the `replicate()` routine. The replication involves only data items associated with the key identifier seven (i.e., $KI = 7$) since only they are encompassed by the specified interval bounds. Thereby, all replicas, associated to this key identifier $KI = 7$, have to be copied so as to restore the whole equivalence class. Precisely, only the data items $di_1 = \langle K1, \Phi \rangle$ and $di_2 = \langle K5, \Phi \rangle$ are replicated to node C.

4. Once the replication message arrives at node C the items are locally stored, albeit at the corresponding key identifier and replication iterator, i.e., $C.get_s(2,1) = \langle K5, \Phi \rangle$ and $C.get_s(2,3) = \langle K1, \Phi \rangle$. Hence, the replication factor $r = 3$ is restored with just one message without affecting the other nodes in the environment.

Finally, the symmetric replication scheme also facilitates load balancing among replicas by sending the `obtainInterval()` request to the least loaded replica, in case metadata on replicas is available. Moreover, an opportunistic approach to load balancing can be taken by sending the `obtainInterval()` request concurrently to all replicas and applying the data from the fastest responding one. Concurrent `obtainInterval()` request generally improve the replication factor recovery behavior, which can to some extent be used for security attack prevention by applying an consensus protocol on the received data items. In any case, the symmetric replication scheme features the lowest overhead, in terms to replication factor maintenance, to traditional approaches.

3.3 Data Consistency

Replication of data items across a set of replication nodes facilitates the availability of data in the event of a failure. In order to replicate the data items at the various nodes

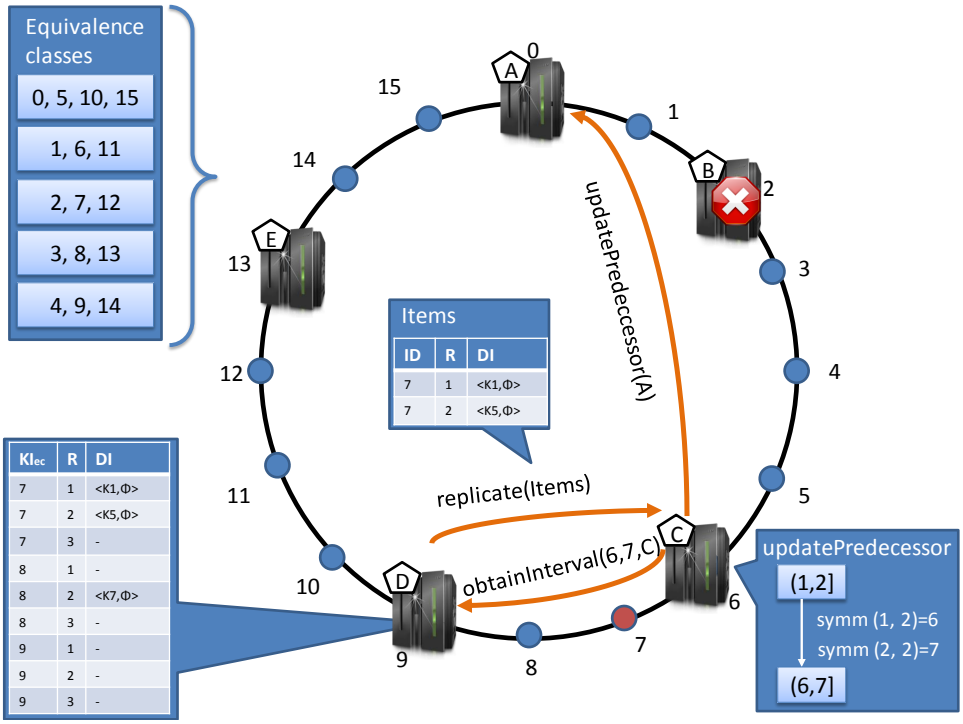


Figure 3.4: Execution example of replication factor restoration

they have to be sent through the transmission network of an execution environment. It takes a small amount of time until all replicated data items physically end up at the destination nodes. Thereby, the exact duration of the replication process depends on the transmission characteristics of the underlying node network. In a static execution environment, where the transmission characteristics are stable and predictable, we assume the replication process to be instantaneous and unproblematic.

Given an execution environment of high dynamics then inherent characteristics of a distributed system such as node overload, failures etc. might severely affect the replication process. In case the transmission network features overloads the replication process prolongs itself due to delayed data items. This implies, that from a temporal point of view not all replica nodes will always feature the same data items w.r.t. their actual values. Simply put, at a specific point in time some replica nodes received the latest copy of the data item subject to replication whereas other replica nodes haven't yet, but will eventually.

To make matters worse, if a subsequent replication process is started before the previous has completed, i.e., has allocated all data items at all replica nodes, the two processes might interfere with each other. In face of network overloads the order of subsequent replication processes might not be fully preserved. Data items might end up at replica nodes in a random order such that some nodes might feature data item values of a previous replication process whereas other nodes might feature data item values of a later replication process.

The distributed data management system thus finds itself in a vulnerable situation in the meantime. It cannot fully count on data availability until all replica nodes have

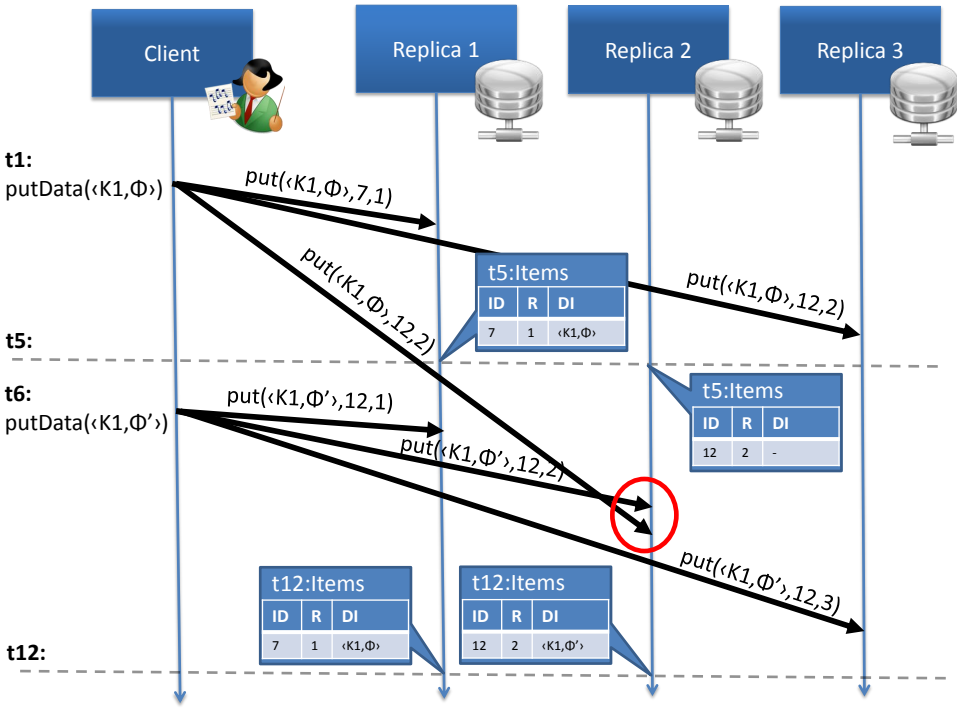


Figure 3.5: Inconsistent data replication given two subsequent replication processes.

received their physical copies. In case it does, it might find out that either data items are missing from some replica nodes, or that they are featuring different values as compared to the other nodes leaving it clueless which data items to use. Such situations lead to a reduced robustness of the system and to a corrupted (inconsistent) state of data from an end-user application point of view.

Example 3.5

Figure 3.5 illustrates the data replication problems, in terms of data inconsistency, from a temporal aspect. As shown the replication process started by the client at timestamp t_1 results in an inconsistent state of the data item at timestamp t_5 across the replicas 1, 2 and 3. Precisely, at t_5 the data item $\langle K1, \Phi \rangle$ is has not yet been replicated to replica 2, unlike replicas 1 and 3.

Moreover, a second replication process is started at timestamp t_5 by the same client on the same data item that forestalls the first replication process, regarding the arrival of data items at replica 2. As indicated by the red circle the updated data item $\langle K1, \Phi' \rangle$ of the second replication process arrives before the data item of the first replication process at replica 2. As a consequence, it gets overwritten with the by now outdated version of the data item of once $\langle K1, \Phi \rangle$ arrives at replica 2. By timestamp t_{12} all replicated data items have arrived at all replica nodes, however in an unequal order. Hence, the replica 1 and replica 2 are inconsistent as they are featuring different data values, i.e., $\langle K1, \Phi \rangle$ and $\langle K1, \Phi' \rangle$ for that timestamp.

For most application scenarios that are relying on distributed data management systems such behavior can severely affect the execution semantics in unforeseeable ways. To prevent situations in which data ends up featuring different values at different nodes the replication process needs to be designed for consistency in first place. The big idea behind data consistency is to serialize the data replication process in a way so that all replica nodes feature the same data item values at all times. Any end-user should perceive the distributed management of the data as if there were just one copy of each data item inside the system. This is referred to as *One-Copy Serializability* [BG85] of the data replication process. When it comes to changing the data, One-Copy Serializability implies that the replication process is conducted in an eager fashion to all replica nodes as a whole or not at all. That is, distributed data items change their values from one consistent version to the next by serializing the replication operations in a way which ensures consistency of all replicas. Formally, data consistency can be described as follows:

Corollary 3.12. *Let T be the set of timestamps $T \subset I\!R^+$ and DHT_s the symmetric replication distributed hash table. The version history of the distributed hash table is defined as the relation $VH \subseteq T \times DHT_s$. A DHT system is consistent if in VH there is not a pair of nodes that features different values of the same data item at the same time:*

$$\forall(t,((di,n,ki),ec,FT(n))) \in VH, \nexists(t,((di',n',ki'),ec,FT(n'))) \in VH :$$
$$di.k = di'.k \wedge di.d \neq di'.d$$

□

Note, that the previous corollary also implies the two different nodes n and n' to feature different Finger Tables and different key identifiers w.r.t. the distributed data item allocation inside the DHT:

$$n \neq n' \wedge ki \neq ki' \wedge FT(n) \neq FT(n')$$

3.3.1 Transaction Management

In order to ensure consistency of data, the transaction model can be exploited. In a nutshell, transactions are logical data handling operations, provided by the applied data management system, so as to guarantee *ACID* [GR92] properties of stored data. *ACID* stands for the *atomicity, consistency, isolation* and *durability* of data that is subject to a series of update operations. Thereby, the data can be touched by a set of transactions that are executed sequentially or concurrently at runtime.

To handle concurrency transactional data management systems rely on concurrency control mechanisms. Based on the read/write model [Vos09], a concurrency control mechanism implies a set of read/write operations (r/w operations) stemming from concurrent transactions on the same data item that have to be ordered (i.e., serialized) at runtime such that the ACID properties are preserved. In case the read/write operations of concurrent transactions cannot be ordered we say that they are in a conflict. Any two transactions are in conflict if they simultaneously feature r/w, w/r, w/w operation intents on the same data item. Along the lines of ACID conflicting transactions have to

be canceled and their effects on the data reverted. To this end our consistency model extends a data item that is subject to transactional handling with the operation (i.e., a read or a write intent) by means of a transaction item which is defined as follows:

Definition 3.13 (Transaction item – item)**.** *A data item, subject to a transactional handling is defined as tuple item* $= (di, op)$ *where* $di \in DI$ *is the affected data item and* $op \in \{R, W\}$ *is the transactional operation.* □

Note, that an read operation is denoted as $op = R$, whereas a write operation is denoted as $op = W$ in an *item* tuple. All transactions of a system are locally checked for validity w.r.t. ACID for each data item operation by the concurrency control mechanism. In case the affected data items are spanning multiple nodes, One-copy serializability of their values has to enforced by means of *distributed transaction* [WV01] protocols.

3.3.2 Distributed Transactions

To extend the read/write model beyond one node only each distributed data item has to be supported by a transaction, i.e., a concurrency control mechanisms, at the corresponding node. Although local transactional handling can enforce the ACID properties at the redundant nodes, from a global point of view this is not any more the case. Especially the isolation property of the replicated data items is difficult to uphold. For instance, a concurrency control mechanism might serialize the r/w operations w.r.t. ACID such that is not conflicting locally but is globally. To isolate the distributed data items on the global level concurrency control mechanisms at the redundant nodes need to be coordinated. Thereby, distributed transaction coordination protocols are built on top of the existing replication scheme so as to coordinate the execution of the distributed concurrency control mechanisms, in terms of r/w operation serialization.

Depending on the actual replication scheme of the underlying system, which can be either synchronous (i.e., eager) or asynchronous (i.e., lazy), global serializability among concurrency control mechanisms can be achieved in two different flavors. Either global serializability conflicts are prevented or they are reconciled. A typical approach to preemptive conflict prevention is pessimistic locking. This approach implies the locking of all data items prior to the start of a distributed transaction upon which the updates can be made. In case a transaction cannot obtain locks on all distributed data items it has to wait and retry. Synchronous replication systems that feature a high conflict probability usually imply pessimistic locking.

On the other hand, asynchronous replication systems usually apply optimistic serializability techniques which implement conflicts detection and reconciliation among the concurrent transactions. To detect conflicts optimistic serializability techniques rely on aids such as transaction timestamps which help to identify the transaction out of the set of concurrent transactions with the latest updates. Reconciliation of conflicting transactions is achieved by applying the operations of the latest one, whereas the outdated ones have to be aborted.

Given an end-user application of the underlying data management system that features transactions of lower conflict probability, the optimistic serializability technique has the upper hand. That is, it allows concurrent transactions to execute at the same

time and thus also favors higher throughput performance of the system. Formally, we can define a distributed transaction as follows:

Definition 3.14 (Distributed transaction – dtx)**.** *A distributed transaction dtx is defined as tuple $dtx = (id, ts, Item, \prec_{op}, alg)$ where:*

- *$id \in K$ is the distributed transaction key by which it is uniquely identified,*
- *$ts \in I\!N^{+} \cup \{0\}$ is the distributed transaction timestamp,*
- *$Item$ is a finite set of transaction items that additionally contains a transaction termination marker τ_{ac} with:*

$$Item = \{item_1, item_2, ..., item_i\} \cup \tau_{ac} \textbf{ with } i \in I\!N^{+}, \tau_{ac} \in \{A, C\}$$

- *$\prec_{op}$ is the partial precedence order relation among the transactional items with:*

$$\prec_{op} \subseteq (Item \times Item)$$

- *alg is the distributed transaction coordination algorithm with:*

$$alg \in \{2PC, PAXOS\}$$

The set of all distributed transactions of a data management system is defined as DTX. □

Note, that global serializability is validated by consistently assessing the precedence order $\prec_{op}$ of the transaction item for conflicts at all redundant nodes. A commit transaction termination marker is denoted as $C \in Items$, whereas an abort transaction termination marker is denoted as $A \in Items$ in an distributed transaction $Items$ set.

To enforce consistency of its data items, w.r.t. One-copy serializability, a distributed data management system, such as our symmetric replication based Chord DHT, has to apply a series of distributed transactions. To this end, we formally define the transactional distributed hash table as follows:

Definition 3.15 (Transactional DHT – DHT_{tx})**.** *Given a symmetric replication distributed hash table DHT_s and the distributed transaction set DTX. A transactional distributed hash table is defined as the relation $DHT_{tx} \subseteq DHT_s \times DTX$ where:*

$$\forall((di, n, ki), ec, FT(n)) \in DHT_s \; \exists dx \in DTX : (di, W) \in dx.Items \wedge C \in dx.Items$$

□

Although distributed transactions guarantee One-copy serializability of redundant data items they also incur additional overhead on the distributed data management system w.r.t. coordination algorithm. We distinguish between more efficient ones over more reliable ones. The remainder of this section discusses at first the most efficient distributed transaction protocol – 2PC – and concludes it with the most reliable one – Paxos.

Two Phase Commit – 2PC

The most basic and the most widely utilized protocol to distributed transactions coordination is the *two phase commit – 2PC* [WV01]. 2PC seeks to prevent globally inconsistent values of redundant data items, which can occur due to lack of coordination among the redundant concurrency control mechanisms. For instance, some replica sites could be committing their r/w operations to the local data management systems while others could be aborting due to some local replica site issues. 2PC is an atomic commitment protocol which enforces all redundantly started distributed transactions to finish their execution in common by either committing or aborting their r/w operations.

To coordinate the distributed transactions 2PC introduces a *transaction manager* that interacts with all replica sites – also referred to as *participants* – in two phases, i.e., in the *voting phase* and the *commit phase*:

- *Voting phase* – In the first phase the manager queries all participants, for their readiness to commit their ongoing transactions. That is, the manager sends a *PREPARE* request to all participants on which they have to respond with a *YES* or *NO* vote. In case the queried participant has locally taken the necessary step to commit its operations the vote is *YES*. In case the queried participant has encountered replica local issues and cannot commit its operation the vote is *NO*.

- *Commit phase* – In the second phase the manager collects all participant votes and assesses them for so as to coordinate all participants as to act as an atomic unit. In case there are no *NO* votes, the manager instructs the participants to commit their operations with a *COMMIT* request. In case there is at least one *NO* vote in the replies all participants have to abort their operations. For this purpose the the manager sends to the participants an *ABORT* request. Upon reception of a *COMMIT* or *ABORT* request all nodes will take the necessary steps to comply with the manager request, i.e., commit or abort the local transactions. Once the local transactions are finished the participants acknowledge the manager by sending an *ACK* message.

The 2PC protocol is illustrated in Figures 3.6 and 3.7 by means of a message flow sequence diagram. Moreover, the Algorithms 3.15, 3.16 and 3.17 show the 2PC coordination message exchange routines from an manager and participant point of view. As we can observe, Algorithm 3.15 shows both phases of the transactional manager coordination. It starts by querying all participants for their local transaction readiness status. First phase responses, i.e., votes, have to be collected till all of them have arrived. Once all participants have cast their vote, and thus their local transaction commit readiness, an assessment can be made whether the distributed transaction is to be committed or aborted. Second phase responses have to be collected as well, so as to conclude whether all participants have successfully concluded their local transactions and released locked resources. Once this occurs the client call that started the distributed transaction, by means of the `commit()` routine, can be informed about the outcome.

Based on the corresponding phase request participants either reply to the manager with the readiness status of the transaction or carry out into execution the manager request by committing/aborting the local transaction. Each readiness response in the first

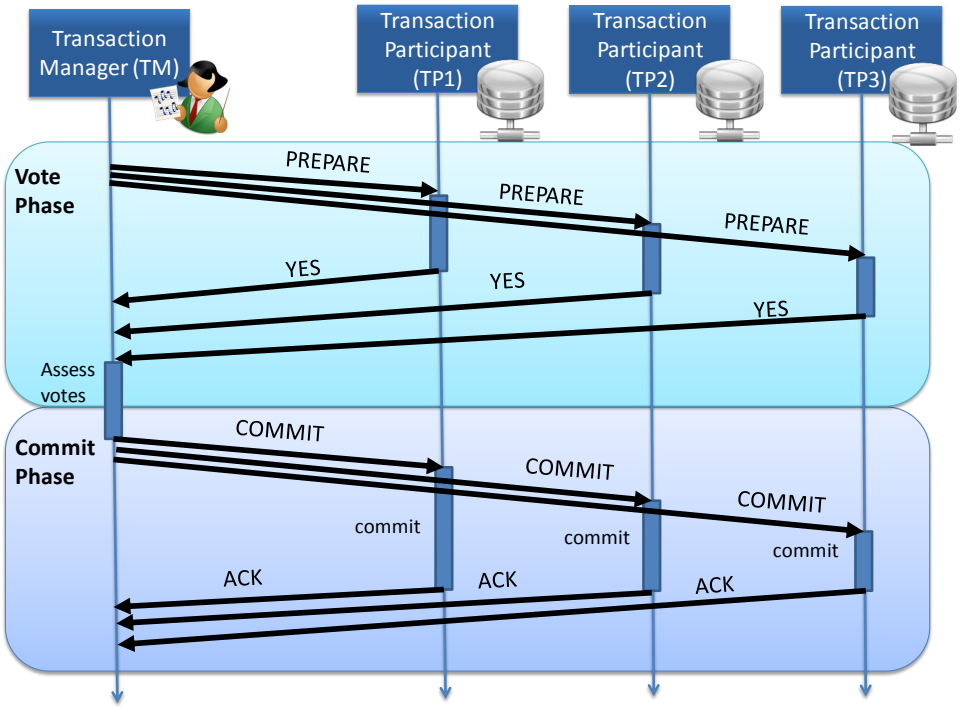


Figure 3.6: Message flow sequence diagram of a successful 2PC commit among one transaction manager (TM) and three transaction participants (TP).

phase is preceded by a locking of the local transactions, i.e., the database resources managed by the local transaction. This way concurrency control among various distributed transactions is facilitated w.r.t. pessimistic locking. Once a distributed transaction has issued a *COMMIT* or *ABORT* request the locally running transactions are concluded with a response to the manager and any locked away data items are released for other distributed transactions to use.

To prevent conflicting situations in which concurrent transactions interfere with each other data management operations, at runtime the participant has to be locked away by the concurrency control mechanism. Precisely, each the encompassed data items by the distributed transaction has to simultaneously be locked away. Even if one data item at one replica site cannot be locked the isolation property could potentially be violated and the distributed transaction has to abort. This way serializability conflicts can be prevented and global serializability preserved among concurrent transactions. In practice 2PC is leveraged in conjunction with the *strict two phase locking* protocol – SS2PL [WV01]. Note, that the method `lock()` of Algorithm 3.16 performs the locking of all data items affected by the transaction whereas the `isCommitReady()` checks for conflicts w.r.t. the serializability of transactions.

Note, that the routines `abortLocal(dtx)` and `commitLocal(dtx)` of Algorithm 3.17 release the locally held locks for the given distributed transaction *dtx*. Moreover, it locally commits (i.e., writes to local storage) the data items comprised by the distributed transaction.

Algorithm 3.15 *tm.commit(dtx)* starts the coordination of the given distributed transaction dtx at manager tm by sending the *prepare* request to each participant tp. Collects all votes and in case there is a NO vote in all participant votes set $Votes$, an $ABORT$ decision is concluded, otherwise a $COMMIT$ decision is concluded. Afterwards it sends the decision to all participants. This routine also collects all acknowledgments from the participants and in case there is no $NOACK$ response from the participants the outcome of the transaction will be $COMMIT$. Otherwise the outcome of the transaction will be $ABORT$. Finally, the outcome is returned to the caller.

```
1:  Votes, AckVotes ← ∅
2:  for all tp ∈ Participants do
3:      Votes ∪ tp.prepare(dtx)
4:  end for
5:  if NO ∈ Votes then
6:      decision ← ABORT
7:  else
8:      decision ← COMMIT
9:  end if
10: for all tp ∈ Participants do
11:     AckVotes ∪ tp.decided(decision, dtx)
12: end for
13: if NOACK ∈ AckVotes then
14:     outcome ← ABORT
15: else
16:     outcome ← COMMIT
17: end if
18: return outcome
```

Algorithm 3.16 *tp.prepare(dtx)* Returns a YES vote at the participant tp in case the given distributed transaction dtx can be locally locked and is ready for a commit. Otherwise it returns a NO vote.

```
1: if tp.lock(dtx) ∧ tp.isCommitReady(dtx) = true then
2:     return YES
3: else
4:     return NO
5: end if
```

The elaborated 2PC protocol offers efficient performances, in terms of distributed transaction coordination, for normal execution scenarios that are free of extreme distributed transaction concurrency and node failures.

In case there are replica node failures in the environment 2PC based system are severely affected as well. Depending on when the node failure occurs 2PC might find itself in difficult situations. To mention some of the possible node failure scenarios, for instance, a participant node could fail during the first phase. This kind of node failure causes the corresponding manager to wait for all votes indefinitely (i.e.,, in Algorithm 3.15 line 5 never occurs). During this time the data items at the running redundant

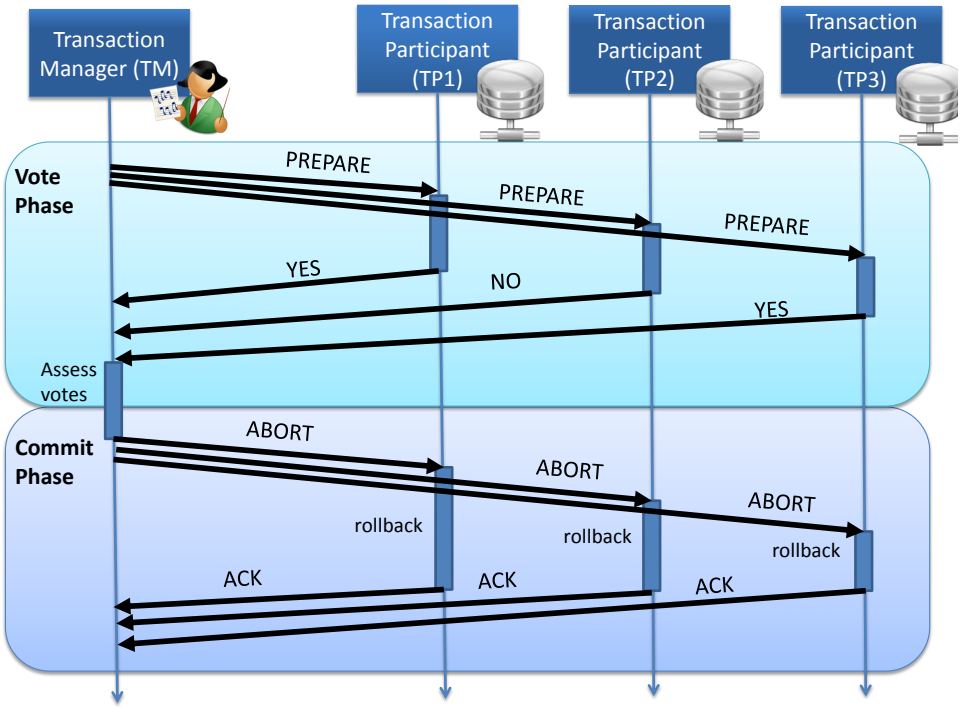


Figure 3.7: Message flow sequence diagram of an unsuccessful 2PC commit among one transaction manager (TM) and three transaction participants (TP).

Algorithm 3.17 *tp.decided(decision, dtx)* Commits and returns a ACK vote at the participant tp in case the given distributed transaction dtx is ready for a commit and the given *decision* is $COMMIT$. Otherwise it aborts dtx and returns a $NOACK$ vote.

```
1: if decision = COMMIT ∧ tp.isCommitReady(dtx) then
2:     tp.commitLocal(dtx)
3:     return ACK
4: else
5:     tx.abortLocal(dtx)
6:     return NOACK
7: end if
```

replica sites are locked away from other transactions and cannot be used. Even worse, if the manager fails after after having issued the $PREPARE$ requests to the participants in any phase the system remains locked away as well. In case the manager fails, there is no one to coordinate the participants so as to request them to release the locally held data item locks (i.e., in Algorithm 3.15 lines 10 – 12 never occur).

To cope with node failures, 2PC relies on message logging in conjunction with time-out detections. Precisely, any incoming message is always logged at all nodes to local stable storage, so that in the event of a failed node recovery the handling of the saved message can be repeated. Moreover, the coordinator can introduce time-outs on participant node interactions. Any delayed participant response can be treated as a failure and handled with an abort of the distributed transaction. However, the mentioned solutions

to 2PC robustness only help to overcome node failures of temporary nature. In general, for 2PC to flawlessly function the assumption of temporary node failures, in conjunction failure incorruptible stable storage has to be made. The permanent manager failure nevertheless prevails as the main disadvantage of the 2PC. This is why it is widely accepted in literature to be a *blocking* distributed transaction coordination protocol.

Paxos

The application of distributed transaction protocols to peer-to-peer environments cannot be based on the assumption of temporary node failures. High dynamics (i.e., scalability) of peer-to-peer networks suggests the nodes to frequently join/leave the execution environment, thus severely affecting ongoing 2PC distributed transactions. In particular the departure of transaction managers, e.g., due to failure, leaves the ongoing transactions blocked and their reserved resources (i.e., data items) locked away.

To cope with transaction manger failures the most straightforward solution is to replicate it across a set of nodes, so that in the event of a temporary or permanent failure the coordination of the distributed transaction can be seamlessly taken over by a substitution manager. To facilitate seamless recovery of failed managers, all involved nodes have to be coordinated in a distributed fashion so as to reach common agreement (i.e., consensus) on the current state of the ongoing transaction. One solution to distributed consensus, in terms of the transaction state agreement, lies in the Paxos atomic commit [GL06] protocol. As the name suggests this protocol is a member of the Paxos [Lam98] distributed consensus protocols family specially designed for distributed transactions.

Basic Paxos

The main idea behind Paxos is to facilitate agreement on a value among a set of participants of a distributed system that is resilient to failures (i.e., non-blocking) and consistent. In Paxos, the participants cooperate, by exchanging messages, so as to choose a common value among themselves. Thereby, the Paxos consensus protocol features a guaranteed robustness to failures of a factor F, such that F stands for the number of failed participants or the number lost (delayed) messages among them. To guarantee fault tolerant consensus of F Paxos makes the following assumptions:

- The involvement of $2F + 1$ participants to a consensus agreement process,
- Reliable communication channels among participants for the exchange of messages,
- Uncorrupted[1] messaging among participants,
- Existence of an eventually perfect fault detector [CGR11] at all participants,
- Existence of an eventual leader detector [CGR11] at all participants.

[1]Delivery of corrupted (i.e., malignantly altered) messages by some of the participants are referred to as Byzantine faults and necessitate the involvement of $2F$ participants.

Hence, Paxos can succeed in reaching a consensus among the participants if the majority of them is alive (i.e., F out of $2F + 1$), if the reliably delivered messages have not been altered (although they could have been delayed or duplicated) and that any failure is detected by means of the eventually perfect fault detector. Thereby, the value that is subject to agreement will be consistent among all participants.

The fault detector at a participant is in charge of periodically probing the other participants by using heart beat messages. Induced by an unresponsiveness alert of the fault detector, a suspected participant is considered to have failed. In case of a delayed response by the suspected participant, the fault detector eventually revises its suspicion and adjusts is unresponsiveness alert threshold with the latest derived value.

The eventual leader detector determines the leading participant among the set of participants. In particular in the case of a node failure, when a new leader has to be elected. Leader election is based on *extrema finding* in which an extreme value of a participant attribute is chosen for the selection of the leader. For instance, the highest participant key identifier space mapping value can be chosen for the selection of the leader. Note, that determining a common leader among a set of nodes is a distributed consensus problem itself that is only a part of the Paxos goal, i.e., the agreement on only one value.

In a nutshell, the consensus protocol of Paxos is based on a series of ballots that are started by participants who want to have their values asserted to others. Thereby, the participant that starts a ballot tries to find a majority of other participants that are willing to accept its value in which case Paxos mandates this value to be consistently asserted as the common value to all participants. To distinguish among specific roles of a ballot Paxos separates all participants into the *proposers*, the *acceptors* and the *learners* as follows:

- *Proposer* – A proposer starts the ballot by proposing a value that is subject to consensus. A proposer is also referred to as the leader of the ballot. The proposer itself is a predetermined acceptor, that is usually selected by a client so as to start the ballot.

- *Acceptors* – Acceptors are the backups of the proposer. They either accept or reject an incoming proposal based on the proposals received so far. Each ballot features a set of $2F + 1$ acceptors out of which at most F can fail. To reach consensus on some value among all acceptors a majority of $F + 1$ acceptors have to be alive and accept the same proposed value.

- *Learners* – Store the accepted outcome of the ballot. That is, learners are the replication sites.

Note, that Paxos allows for the combination of different roles at the same node. For instance, a node might be an acceptor and a learner at the same time.

In general, a Paxos ballot consists of two main phases, a *read* phase and *write* phase. Each of this phases is further divided into additional sub-phases. In the read phase a Paxos proposer queries a quorum of acceptors for its readiness to accept its proposed value. Based on the responses of the acceptors from the read phase, that might feature

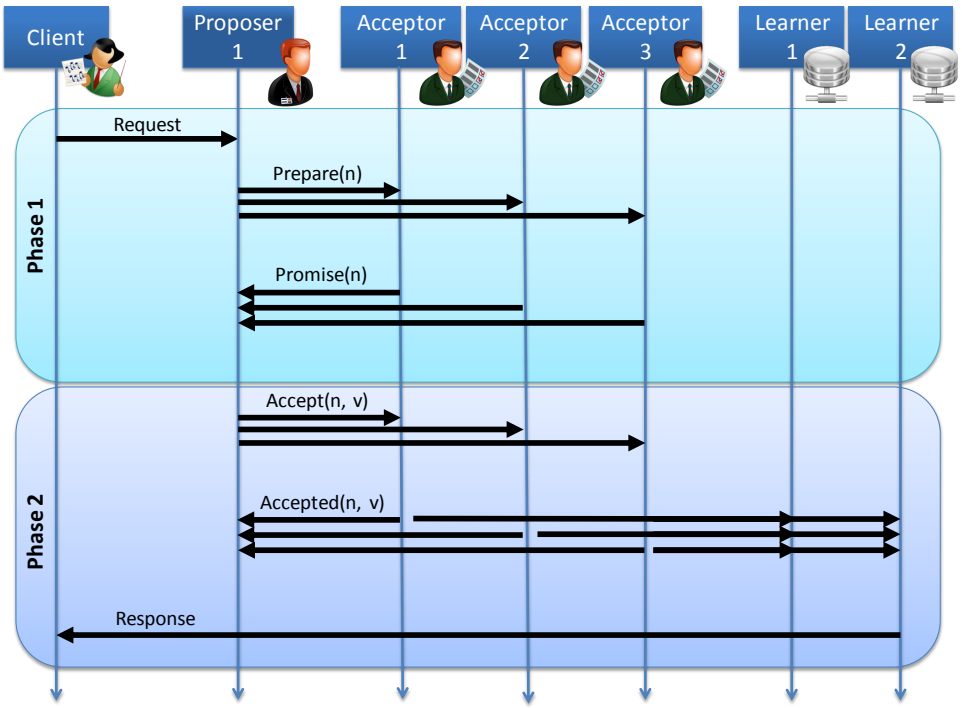


Figure 3.8: The basic message flow sequence diagram of a successful Paxos consensus among one proposer, the acceptors and the learners induced by client requests.

a new value to be accepted, in the write phase of Paxos the proposer tries to enforce a value to the acceptors. The two ballot phases of Paxos are conducted as follows:

- *Phase* $1a$ *– Prepare*. The proposer p sends a $PREPARE$ message to all acceptors in conjunction with a new ballot number n. The ballot number n must be greater than any other ballot number encountered so far by this proposer.
- *Phase* $1b$ *– Promise*. The acceptor a receives the $PREPARE$ message and promises to accept the proposal n if it is the highest ballot number encountered so far. Otherwise, it rejects the proposal with a $NACK$ message. In conjunction to the $PROMISE/NACK$ message each acceptor replies with the highest ballot number encountered so far and its corresponding state, i.e., the previously accepted value v_a and its associated ballot number n_a.
- *Phase* $2a$ *– Accept*. Upon reception of a majority of acceptor $PROMISE$ messages, the proposer enforces a value v with the ballot number n to the acceptors by means of an $ACCEPT$ message. Thereby, the value v corresponds to the initial value of the proposer (received from a client) or the highest ballot number value v_x received as the replied acceptor state. The client value may only be sent if no acceptor states values are received that have been previously accepted.
- *Phase* $2b$ *– Accepted*. The acceptor acknowledges the proposer value v and the ballot number n by returning an $ACCEPTED$ message featuring n and v. However, in case another proposal with a higher ballot number (i.e., greater than n)

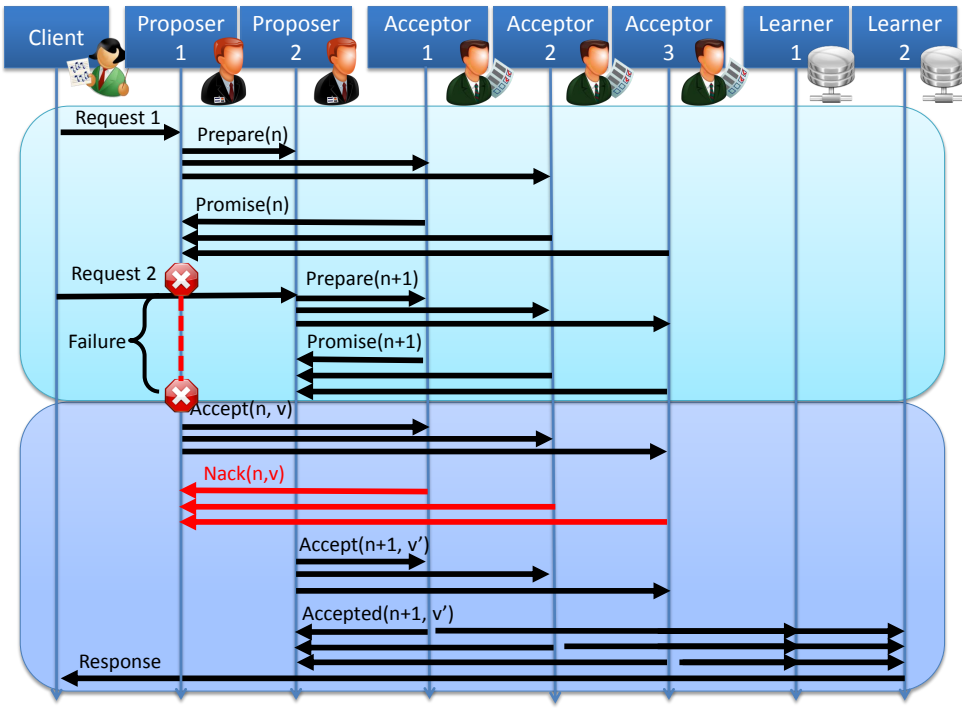


Figure 3.9: The basic message flow sequence diagram among one proposers, the acceptors and the learners induced by client requests. Successful Paxos consensus for proposer 2, unsuccessful consensus for Proposer 1

has been issued to the acceptor in the meantime, the current *ACCEPT* request, in terms of v and n, is rejected by means of a *NACK* message. Is such a scenario the ballot of p starts all over with a higher ballot number n'.

Note, that basic Paxos features in literature a third phase that consists of sending the accepted value, by the majority of acceptors, from the proposer to the learners. This process can however be combined with the *ACCEPTED* message of *Phase* 2*b* so as to improve the performance of Paxos by reducing the number of message delays.

The complete Paxos message flow among the participants is illustrated in the Figures 3.8 and 3.9. Note, that the acceptors are not sending any state back, which allows the proposer to enforce its value v onto them (e.g., Acceptor 1 of Figure 3.8). In case any of the acceptors fails, consensus can nevertheless be reached and the learners informed if the remaining acceptors manage to cast their votes to the proposer. In case the proposer fails (e.g., Proposer 1 of Figure 3.9) a substitution proposer with a new Paxos instance can be elected, by means of the leader detector, that features a new value and a higher ballot number (e.g., $n + 1$) as compared to the previous proposer. If failed proposer returns to enforce its value to the acceptors, it will find its proposal out-dated (i.e., the ballot number lower compared to the actual one), by means of acceptor rejections, in which case it can try to start all over with a new ballot. A drawback of the Paxos consensus approach lies in the fact that concurrent proposers can find themselves in mutually overbidding ballot number deadlock which can drag out over many rounds.

Many siblings in literature [LM04, Lam06, Lam] exist aiming to optimize and overcome the drawbacks of Paxos.

Paxos Atomic Commit

Due to its two phases, Paxos bears a certain resemblance to the 2PC protocol. In fact, the two phases of the 2PC protocol can be easily fused with Paxos consensus features so as to facilitate fault-tolerant coordination of distributed transactions. In a nutshell, 2PC distributed transactions can be made impervious to blocking transaction manager failure if the transactional manager is replicated across a set of backup nodes. This set of redundant transaction managers is subject to Paxos consensus agreement on the outcome of the transaction.

Precisely, the redundant transactional manager set is mapped to the Paxos acceptors set. The initial Paxos leader (i.e., proposer) is thus the 2PC transactional manager that is issued with the client request so as to coordinate the distributed transaction. Consequently, the value that is subject to agreement among the redundant transaction managers (i.e., acceptors) is the state of the transaction, i.e., the COMMIT/ABORT outcome. In case the leader fails, after having sent out the $PREPARE$ messages of 2PC, one of the redundant managers can seamlessly take over the coordination of the distributed transaction as they all share the same transaction state. In such a scenario the substitution transaction manager can send to the participants a $COMMIT/ABORT$ decision and this way prevent blocking.

To facilitate seamless transfer of leadership each transaction manager redundantly runs an instance of the Paxos protocol with itself as the leader, and all transaction managers as the acceptor set. This way a substitution manager can start a new Paxos ballot on the same managers set whenever it suspects the initial leader to have failed and try to enforce its view of the transaction state on the others. Of course, Paxos guarantees that only one consistent transaction state will exist among the redundant transaction managers. Thereby, consensus can be reached if a majority of the redundant transaction managers (i.e., $F + 1$ acceptors) remains alive. The new leader is elected by means of the eventual leader detector whenever the initial leader failure occurs.

The transactional participants of 2PC, that locally manage the data items, correspond to the learners of the Paxos ballot. However, they do not just learn the outcome of the transaction but they also provide the transactional managers with initial state of the transaction (i.e., the commit readiness status) which is subject to Paxos agreement. Hence, an atomic commit of the distributed transaction is only possible, from a Paxos enhanced 2PC point of view, if:

1. All transactional participants remain alive and are ready to commit,
2. A majority of transactional managers has agreed on the participant readiness state and have this acknowledged by the leader.

To apply Paxos on top of 2PC a series of message exchange phases among all participants is necessary. Before the transaction can even start, the leader needs to initialize its acceptor set, i.e., the redundant transaction managers. Once the acceptors have been initialized the initial transaction state needs to be retrieved from the learners, i.e., the

transactional participants. Only after that, the Paxos ballot can be started so as to agree on the same value at the acceptors and to decide on the outcome of the transaction. A Paxos atomic commit execution is conducted as follows:

1. *commit* – A client issues the $COMMIT$ request for a $Item$ set to a transaction manager that subsequently becomes the leader of the consensus.

2. *initPaxos* – The leader selects its set of replication transactional managers TM and initializes instances of Paxos at them together with the $Item$ set that is subject to the commit. Initialization of the acceptors by the leader implies that their consensus state is set to $NULL$ such that a $Phase\ 1a(Prepare)$ message would always result in an empty state $Phase\ 1b(Promise)$ message. Hence, by initializing the managers (i.e., the acceptor set) itself the leader does not need to query the acceptors for their state with a $Phase\ 1a(Prepare)$ message and the first (i.e., read) phase of Paxos can safely be omitted.

3. *registerPaxos* – The redundant managers and their Paxos instances acknowledge their participation to the leader as acceptors for the Paxos consensus.

4. *setTM* – Once aware of all transactional managers TM, the leader further initializes all Paxos instances at the managers with the acceptor set TM and learner set TP so as to facilitate recovery in case of its failure.

5. *prepare* – Now that the leader has been replicated across the acceptor set the distributed transaction coordination can commence. For this purpose, leader has to query the consensus value, i.e., the commit readiness status of the learners TPs, for the set $Item$. Hence, the leader sends a $PREPARE$ message containing $Item$ and the ballot number to the learner set TP.

6. *vote* – Depending on whether they have locally taken the necessary steps to commit the $Item$ set, learners have to respond with a ($COMMIT$ or $ABORT$) vote. The resulting vote can be submitted to the leader itself in which case the leader needs to aggregate all votes and send them to the acceptor set TM as a $Phase\ 2a(Accept)$ messages. However, in order to improve the performance of Paxos commit the responsibility of sending the votes is delegated directly to learners, thus removing the unnecessary message delay via the leader. Essentially, the learner votes that are sent to the acceptor set represent the Paxos $Phase\ 2a(Accept)$ messages of the leader that are directly sent by the learners.

7. *voteAck* – The acceptors aggregate all learner votes so as to decide on the outcome of the transaction. Once all votes are aggregated and assessed each acceptor sends a $VOTEACK$ message to the leader which essentially corresponds to the $Phase\ 2b(Accepted)$ message of Paxos consensus. Thereby, an acceptor decides to accept the distributed transaction to be commit ready only if all learners returned a $PREPARE$ vote in conjunction with a ballot number that is the highest so far. In case even a single learner returns an $ABORT$ vote, the $VOTEACK$ message of the acceptor will feature an $ABORT$ consensus value.

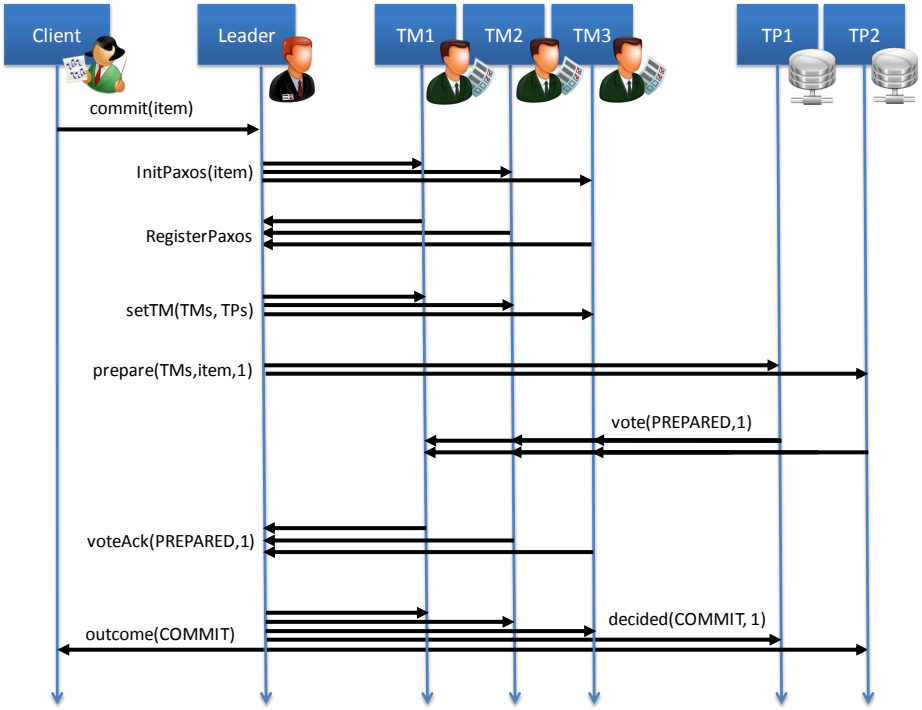


Figure 3.10: The message flow of the Paxos commit protocol

8. *decision/outcome* – Once the leader receives a majority of acceptor *VOTEACK* messages featuring the same value (i.e., *PREPARE* or *ABORT*), a Paxos consensus has been reached among the acceptors and the learner can conclude the transaction. Consequently, all participants of the transaction (i.e., the acceptors, the learners and the client) can be can be informed about the outcome by means of *DECIDED* and *OUTCOME* messages, respectively.

The complete Paxos commit protocol, and its message flow, is illustrated by means of Figure 3.10. Moreover, the Algorithms 3.18 – 3.28 describe in detail the Paxos atomic commit procedure. Note, that the Paxos commit algorithms feature a concurrency control mechanism that implements the optimistic serializability approach. Together with transactional timestamps the elaborated concurrency control mechanism also exploit locks so as to detect conflicting transactions. In face of multiple data items comprised by concurrent transactions of the same timestamp, locks are necessary so as to maintain integrity of an individual transaction between asynchronous replication requests. Precisely, only if a concurrent transaction features the latest timestamp and holds locks for all of its data items it is resolved as a successful transaction. The others are not and thus have to be aborted.

Given the Paxos commit algorithms, the initiation of a Paxos distributed transaction is shown in Algorithm 3.18. In this routine the leading transaction manager starts off the distributed transaction by immediately selecting the acceptors set (i.e., the redundant transaction managers) and initializing them with Paxos instances.

Each acceptor needs to initialize an instance of Paxos locally with the same input parameters as the leader such that seamless recovery of the leader for the current trans-

Algorithm 3.18 *leader.commit(dtx, client)* Given the distributed transaction dtx this routine determines the replication transaction managers and initializes them with same client input parameters: dtx and *client*.

```
1: leader ← this
2: for all item ∈ dtx.Items do
3:     for i ∈ R do
4:         tm_i ← n.getTM(dtx, i)
5:         tm_i.initPaxos(leader, client, item, dtx)
6:     end for
7: end for
```

action is possible. The initialization parameters feature the client (that commissioned the transaction) and the item (that is subject to commit). Along the lines of Paxos consensus, the state of each acceptor needs to be reset. An acceptor reset implies the setting of the ballot number to 1 and transaction state value to *null*. This way any proposal, consisting of a ballot number and transaction state, by the initial leader will be accepted, since it will always be higher than the reset ballot number. As a consequence the Phase 1 message flow of Paxos can be skipped. Upon successful initialization the acceptors have to register themselves with the leader. Algorithm 3.19 shows the initialization of a Paxos commit acceptor transaction manager.

Algorithm 3.19 *tm.initPaxos(l, cl, it, dtx$_r$)* Given the leader l, the client cl the transaction dtx_r and the transaction it it initializes the local Paxos instance at the transaction manager. Subsequently, it resets the ballot numbers and the votes for the data item and registers this instance of Paxos with the leader.

```
1: leader ← l
2: dtx ← dtx_r
3: item ← it
4: client ← cl
5: for i ∈ R do
6:     Votes[item.di.k][i] ← (⊥, 1, 0)
7:     AcksTMs[item.di.k] ← ∅
8: end for
9: leader.registerForPaxos(this)
10: state ← COLLECTING_VOTES
```

Note, that in Algorithm 3.19 a Paxos commit transaction can feature more than just one item. This implies that all items have to be distinguished, and agreed upon by the acceptors. Only when consensus on all items has been reached a transaction be concluded.

Once all acceptors have registered with the leader, the learners, i.e., the transactional participants, need to be determined for this transaction. Learner readiness, with respect to a commit on *item*, has to be determined so as to start the Paxos consensus with the acceptors. To retrieve the commit readiness of the learners for *item* they have to be queried by the leader by means of the *prepare()* routine. Just before the Paxos consensus

can start all acceptors need be additionally initialized with the the registered acceptor set TMs and the newly determined learner set TPs so as to fully replicate the leader and thus facilitate its fault-tolerance. Algorithm 3.19 shows the registration of the acceptor set and the subsequent determination of the learners.

Algorithm 3.20 *leader.registerForPaxos(tm_i)* Aggregates all acceptors based on the registered tm_i. Afterwards, retrieves all participants and queries them for the commit readiness status. Finally, it initializes all acceptors with the acceptors set and the learner set.

```
1: TMs ← tm_i
2: if |TMs| = r then
3:     for all item ∈ dtx.Items do
4:         for i ∈ R do
5:             tp_i ← n.getTP(item, i)
6:             TPs ∪ (tp_i, item, i)
7:             tp_i.prepare(dtx, item, i, TMs)
8:         end for
9:     end for
10:    for tm ∈ TMs do
11:        tm.setTM(TMs, TPs, dtx)
12:    end for
13: end if
```

Each acceptor needs to be additionally initialized with the same parameters, i.e., the complete acceptor set and the complete learner set, as the leader so that seamless recovery of the leader is possible. To facilitate the leader recovery, the eventually perfect fault detector of each acceptor is supplied with all participants so as be able to detect learner failures. Moreover, the leader detectors of each acceptor are supplied with all acceptors so as to elect a new leader in case the initial one fails. Algorithm 3.21 shows the completion of an acceptor initialization.

Algorithm 3.21 *tm.setTM(TM, TP)* Initializes the local fault detector with the given learner set TP and initializes the local leader detector with the given acceptor set TM.

```
1: TMs ← TM
2: TPs ← TP
3: faultDetector.init(TPs, dtx)
4: leaderDetector.init(TMs, dtx)
```

Once a learner receives a commit readiness query from the leader it has to take note of the transactional item. Moreover, it has to perform conflict detection and reconciliation w.r.t. global serializability. The vote of the learner will be based on the outcome of the serialization conflict detection. Subsequently, the learner vote becomes the transaction state that is subject to consensus and it is proposed by the learner as a *Phase* 2*a* Paxos consensus message. Algorithm 3.22 shows the commit readiness query at the learner.

The optimistic serializability technique featured by concurrency control mechanism of the learner is based on versioning timestamps in conjunction to read and write locks.

Algorithm 3.22 *tp.prepare(dtx , item i TMs)* Stores the *item* locally as involved in the transaction dtx. Locks the *item* and based on the success of the locking attempt proposes a transaction state.

1: $TxItems \cup (dtx, item.di.k)$
2: $vote \leftarrow n.serialize(dtx, item, i)$
3: $n.propose(item, i, TMs, vote, 1)$

The read and write locks locally store the operation intents of concurrent transactions for each data item of it. Since we assume an asynchronous replication scheme, with the help of locks, serializability conflicts are effectively detected between asynchronous commit requests of concurrent transactions. Precisely, each data item has to be checked for a lock w.r.t. the operation intent and if no other concurrent transaction has already placed a lock on it, serializability is not affected.

Algorithm 3.23 *tp.serialize((dtx, item, i)* Given the *item* checks whether there are writing locks in *writeLock* on it in which case an already existing lock results in an $ABORT$ vote. If not, the routine checks whether read locks exist in *readLock*. Sets another lock in *readLock* and returns $PREPARED$ for a read operation transaction. Checks a write operation transaction for the timestamp as compared to the locally stored version. In case of a higher timestamped transaction item places a lock in *writeLock* and returns $PREPARED$. Finally, stores all learner votes to local logs.

1: $ki \leftarrow n.getRep(item, i)$
2: $dbItem \leftarrow n.getItem(item, i)$
3: $vote \leftarrow ABORT$
4: **if** $writeLock[(item.di.k, ki)] = 0$ **then**
5: **if** $item.op = WRITE$ **then**
6: **if** $readLock[(item.di.k, ki)] = 0 \wedge dts.ts > dbItem.ts$ **then**
7: $writeLock[(item.di.k, ki)] = 1$
8: $vote \leftarrow PREPARED$
9: **end if**
10: **else**
11: **if** $ts.ts \geq dbItem.ts$ **then**
12: $readLock[(item.di.k, ki)] = readLock[(item.di.k, ki)] + 1$
13: $vote \leftarrow PREPARED$
14: **end if**
15: **end if**
16: **end if**
17: $n.storeToLog(dtx, item, i, ki, vote)$
18: **return** $vote$

In general, the learner will not detect serializability conflicts if it tries to read the data item and if there are no write locks, stemming from concurrent transactions, already placed on it. Likewise the learner will not detect conflicts if it tries to rewrite the data item with a new version (i.e., a higher transaction timestamp) and there are no write locks already placed on it by some other transaction. To this end, each locally

stored data item needs to feature a timestamp of its most recent update (e.g., the *dbItem* data structure). Only in these cases the concurrency control mechanism of the learner approve of the transaction and vote to be $PREPARED$ for a commit. This implies that the approved transaction will place its locks, either on the write lock or on the read lock (depending on the transaction operation), such that concurrent transactions are rejected. For all the other cases serializability conflicts will be detected and the transaction reconciled with a rejection of the commit request.

Moreover, to facilitate learner recovery from temporary node failures all placed locks are permanently persisted to local stable storage, i.e., logs. Algorithm 3.23 shows the concurrency control mechanism of the learner.

Knowing that the reading phase (i.e., *Phase* $1a, 1b$) of Paxos is skipped in the first proposal of the leader the learner can cast its vote directly to all acceptors if it is addressed with a ballot number 1. In such a scenario the learner vote corresponds to a *Phase*2*a* message of the Paxos consensus. In case the ballot is higher than 1 a regular Paxos consensus has been started by some acceptor. In a regular Paxos scenario learner casts its vote as a *Phase*1*a* message to the acceptor set. Algorithm 3.24 shows the vote casting by the learner.

Algorithm 3.24 *tp.propose(*$k, i, TMs, vote, ballot$*)* Casts the learner votes to all acceptors TMs for the item key k and replication enumerator i.

```
1: for tm ∈ TMs do
2:     if ballot = 1 then
3:         tm.vote(k, i, vote, ballot)
4:     else
5:         tm.readVote(k, i, vote, ballot)
6:     end if
7: end for
```

The learner votes are all aggregated by each acceptor so as to derive the proposed value for the consensus. Thereby, each vote has to be checked against its ballot number in accordance with *Phase* $2b(Accepted)$ of Paxos. Only if the received vote is of the highest ballot number encountered so far than it will get accepted by the acceptor, otherwise it will not. When all learner votes have been accepted the value of the consensus can be derived. That is, in case there is an $ABORT$ response among the votes, some learner could not guarantee serializability of the data item, in terms of concurrency control. This means that the transaction needs to be aborted from the acceptor point of view and the consensus value to be accepted is $ABORT$. On the other hand, if there are only $PREPARED$ votes the transaction can be committed by all learners and the consensus value to be accepted becomes $PREPARED$. Once derived, the consensus value of the acceptor has to be acknowledged to the leader by means of the *Phase* $2b(Accepted)$ message. Algorithm 3.25 shows the acknowledgment of the leader votes by the acceptor.

Note, that in the *Votes* data structure of Algorithm 3.25 the read phase and the write phase ballot numbers of Paxos are distinguished. The reason behind this lies in the fact that the leader has to be able to select the vote value of the highest write phase ballot number in *Phase* $1b(Promised)$ of Paxos.

Algorithm 3.25 *tm.vote(k, i, vote, ballot)* Accepts a learner *vote* for a item key *k* if the *ballot* number is higher than any number encountered so far. When all replica votes have been received the routine acknowledges the acceptance to the leader with a *PREPARED* vote in case no *ABORT* have been cast by the learners. Otherwise an *ABORT* is acknowledged to the leader.

```
1: (currVote, rBallot, wBallot) ← Votes[k][i]
2: if ballot ≥ rBallot ∧ ballot ≥ wBallot then
3:     Votes[k][i] ← (vote, rBallot, ballot)
4: end if
5: ReplicaVotes ← ∅
6: for j ∈ R do
7:     ReplicaVotes ∪ Votes[k][j]
8: end for
9: if |ReplicaVotes| = r ∧ ∄rVote ∈ ReplicaVotes : rVote.currVote =⊥ then
10:    if ∄repVote ∈ ReplicaVotes : repVote.vote = ABORT then
11:        tmVote ← PREPARED
12:    else
13:        tmVote ← ABORT
14:    end if
15:    leader.ackVote(k, ballot, tmVote)
16: end if
```

The leader collects all *Phase* 2*b*(*Accepted*) acknowledgment votes stemming from the acceptors. When a majority of acknowledgment votes features the same value (i.e., *ABORT* or *PREPARED*) the leader is be able to conclude the transaction. A majority of *PREPARED* votes acceptors will result in a commit of the transaction, whereas a majority of *ABORT* votes will result in an abort of the transaction. However, the majority vote has to apply for all transactional items. If even a single item features a majority *ABORT* vote outcome, the whole transaction has to be aborted. Algorithm 3.26 shows the acknowledgment of the acceptor votes by the leader.

A decision of the leader has to change its internal state so as not to accept votes any more. Moreover, the outcome decision has to be communicated by the leader to all participations of the transaction, i.e., the acceptors, the learners and the clients. Algorithm 3.27 shows the decision making by the leader.

Note, that the *decided()* routine of a transaction manager is not further considered as it merely sets the internal state to a *DECIDED* value just as in line 1 of Algorithm 3.27.

A decision taken on the learner has however several effects. First any read or write locks, held by this transaction on the data item, have to be released so that the item can be updated in the future. In case, the learner successfully locked the data item away with a writing operation intent and the outcome of the transaction was a *COMMIT* then the data item value has to be persisted to the local data management system for good. Moreover, the data item has to be removed from recovery log and from the transactional item storage.

Algorithm 3.26 *leader.ackVote(k, tmVote, ballot)* Collects acceptor votes *tmVote* for data item key *k* and *ballot* number. In case of a majority of *PREPARED* votes for all items a commit decision is issued, in case of a majority of *ABORT* votes for at least one item then an abort decision is issued by the leader.

```
1: AckTMs[k] ∪ (tmVote, ballot)
2: Preps ← ∅
3: Abs ← ∅
4: ItemOutcomes ← ∅
5: for all item ∈ dtx.Items do
6:     for all prep ∈ AckTMs[k] : prep.vote = PREPARED do
7:         Preps ∪ prep
8:     end for
9:     for all abs ∈ AckTMs[k] : prep.vote = ABORT do
10:        Abs ∪ abs
11:    end for
12:    if |Preps| ≥ ⌊r/2⌋ + 1 then
13:        ItemOutcomes[k] ∪ COMMIT
14:    end if
15:    if |Abs| ≥ ⌊r/2⌋ + 1 then
16:        ItemOutcomes[k] ∪ ABORT
17:    end if
18: end for
19: if |ItemOutcomes| = |dtx.Item| then
20:    if ABORT ∈ ItemOutcomes then
21:        leader.decide(ABORT)
22:    else
23:        leader.decide(COMMIT)
24:    end if
25: end if
```

Algorithm 3.27 *leader.decide(decision)* Changes the leader *state*. Informs all acceptors, all learners and the client about the leader *decision* on the outcome of the transaction.

```
1: state ← DECIDED
2: for tp ∈ TPs do
3:     tp.decided(decision)
4: end for
5: for tm ∈ TMs do
6:     tm.decided(decision)
7: end for
8: client.outcome(decision)
```

In the meantime of the transaction conclusion, nodes might have subscribed to the learner for the outcome of the transaction with regard to the data item. Node outcome subscriptions can happen due to the change of responsibility over the data item by the

underlying data management system. In such an event, whenever the data item changes due to the transaction outcome the subscribed nodes have to be updated, as well. That is, if the data item is not involved any more in any other transaction and if nodes are subscribed to it, then they all have to be updated. Algorithm 3.28 shows the steps taken by the learner when the transaction is concluded.

Algorithm 3.28 *tp.decided(decision)* Releases the write lock or the read lock if any for all transactional items. In case of a commit *decision* this routine persists the *item* to local storage. Afterwards finds all subscribed nodes for *item* and removes them from the *Subscribers* list. Updates the subscribed nodes with *item*, based on the replication enumerator i, if *item* is not involved in any other transaction.

```
1:  StoredItems ← getFromLog()
2:  for all (dtx, item, i, vote) ∈ StoredItems do
3:      ki ← n.getRep(item, i)
4:      if item.vote = PREPARED then
5:          if item.op = WRITE then
6:              writeLock[(item.di.k, ki)] = 0
7:              if decision = COMMIT then
8:                  n.putItem(item, i)
9:              end if
10:         else
11:             readLock[(item.di.k, ki)] = readLock[(item.di.k, ki)] − 1
12:         end if
13:     end if
14:     for all s ∈ {s|(s, st_dtx) ∈ Subscribers[item.di.k] ∧ st_dtx = dtx} do
15:         Subscribers[item.di.k] \ (s, dtx)
16:         if ∄(s_x, t_x) ∈ Subscribers[item.di.k] : s_x = s ∧ t_x ≠ dtx then
17:             s.putItem(item, i)
18:         end if
19:     end for
20:     TxItems \ (dtx, k)
21: end for
22: storeToLog(decision)
```

The considered algorithms so far describe the execution of Paxos commit in a failure-free setting. In case node failures do happen, Paxos commit has to ensure consensus of the transaction outcome among the remaining nodes. Unlike 2PC, Paxos commit should feature failure handling that overcomes blocking of the consensus agreement for any kind of node failure.

Thereby, Paxos commit distinguishes between node roles when performing failure handling. That is, leader failures, acceptor failures, and in particular the failure of the leading acceptor are treated separately. The transactional participants role based failures are handled as follows:

- Learner failure – If a learner fails the transaction has to result in an *ABORT* outcome due to the atomicity requirement of any distributed transaction. Whenever

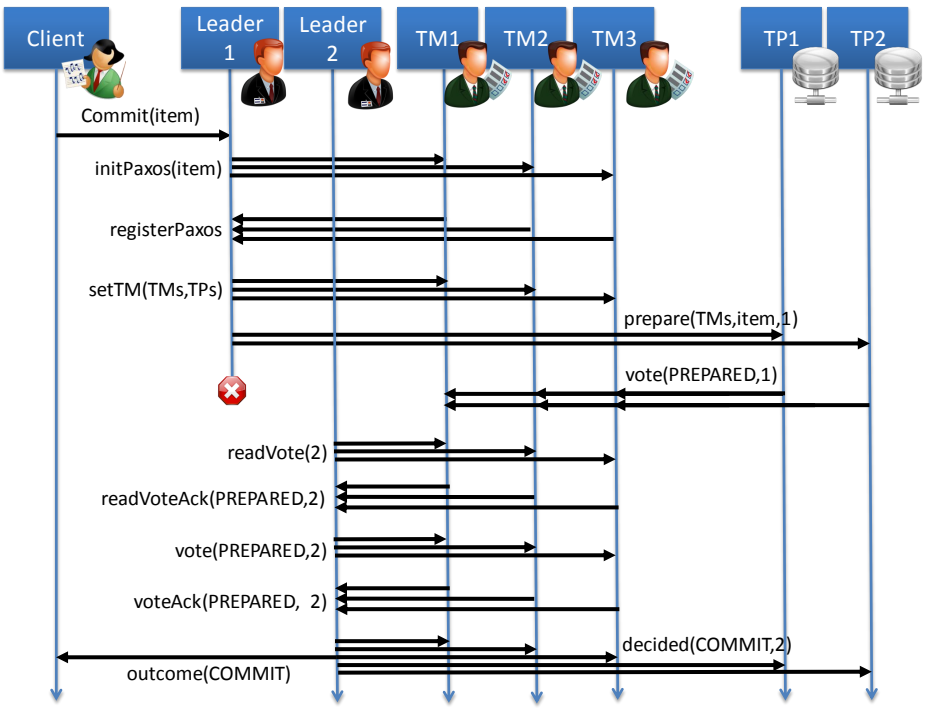


Figure 3.11: The Paxos commit algorithm in the event of a leader failure.

a learner failure is detected, by means of the fault detector, the leader proposes an *ABORT* vote on behalf of the failed learner, by means of a new Paxos ballot, thus causing the transaction to conclude in an *ABORT* outcome.

- Acceptor failure – If an acceptor fails the transaction is not affected if it the failed acceptor is not the leader. That is, as long as a majority of acceptors is alive consensus agreement can be reached and the transaction will eventually conclude.

- Leader failure – If the leader fails a new leader has to be elected, by means of the leader detector. The new leader, has to learn the status of the ongoing transaction by means of a new Paxos ballot. In case a consensus has been reached the new leader can decide on the outcome of the transaction. If no votes have been cast the leader can preemptively abort the transaction.

The described Paxos commit failure handling, that distinguishes among the node roles, is illustrated by means of the Figures 3.11 and 3.12. Moreover, the Algorithms 3.29 – 3.11 describe in detail the Paxos atomic commit routines that are executed whenever a node failure occurs.

The failure of a leader is always detected by the leader detector, upon which a new leader is automatically elected. The newly elected leader starts a new Paxos ballot as it has to learn the current consensus status of the transaction before making a decision. To learn the transaction status the new leader sends a $Phase1a(PREPARE)$ message to all acceptors featuring a new ballot number and no proposed value. Naturally, the new ballot number has to be highest ballot number encountered so far. An usual way of

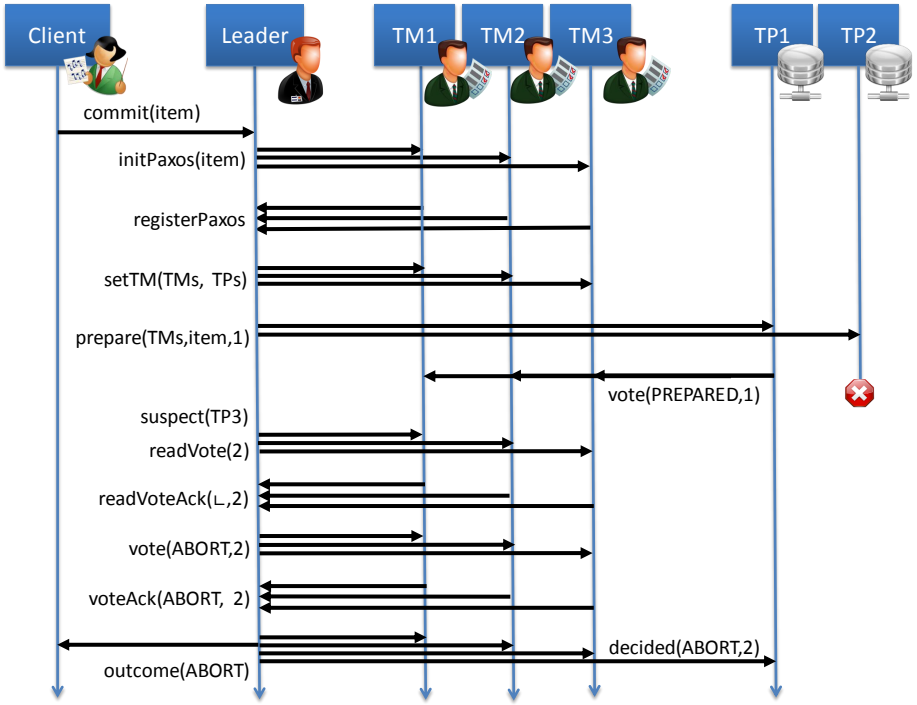


Figure 3.12: The Paxos commit algorithm in the event of a participant failure.

selecting a higher ballot number is to exploit the node key identifier number increased by one, i.e., $f_n(newLeader) + 1$. Precisely, as leader election is based on key identifier number extrema finding the new leader (with the highest key identifier number) can use its key number, however increased, to propose a new ballot. New ballot selection is provided by means of the `nextBallot()` routine. Algorithm 3.29 shows the steps taken by the new leader when elected by the leader detector.

Algorithm 3.29 *tm.trust(dtx, newLeader)* Creates a new ballot number for the given transaction dtx. In case the transaction has not been concluded and this acceptor is the new leader it starts a new ballot.

1: $leader \leftarrow newLeader$
2: $newBallot \leftarrow n.nextBallot(dtx)$
3: **if** $state \neq DECIDED \wedge leader = this$ **then**
4: **for all** $item \in dtx.Item$ **do**
5: **for** $i \in R$ **do**
6: $Votes[item.di.k][i] \leftarrow (\perp, 1, 0)$
7: $n.propose(dtx, item.di.k, i, \perp, newBallot)$
8: **end for**
9: **end for**
10: **end if**

The proposal by the new leader is aggregated by each acceptor as in accordance to Paxos consensus in *Phase*1*a*. In case the proposal features the highest ballot number encountered so far the acceptor forwards a *Phase*1*b* message. Thereby, the *Phase*1*b* mes-

sage features the current transaction state of the previously accepted *Phase*2*b* consensus value and its associated writing ballot number. Algorithm 3.30 shows the acknowledgment of the leader proposal by the acceptor.

Algorithm 3.30 *tm.readVote(k, i, vote, ballot)* Reads the current vote for an item, in terms of key k and replica identifier i, if the *ballot* number is highest so far. Afterwards, acknowledges the current vote to the learner.

```
1: (currVote, rBallot, wBallot) ← Votes[k][i]
2: if ballot > rBallot ∧ ballot > wBallot then
3:     Votes[k][i] ← (currVote, ballot, wBallot)
4:     leader.readVoteAck(k, i, ballot, currVote, wBallot)
5: end if
```

In accordance to *Phase 1b* of Paxos the new leader aggregates all acceptor *PROMISED* (i.e., *readAck*) messages by means of the *ReadVotes* data structure. Upon reception of a majority of acceptor read votes the new leader learns the transaction status value by selecting the read vote of the highest ballot number. It case a consensus has been reached among the acceptors and the new leader can try to enforce the consensus value at all acceptors by means of the second (i.e., write) phase of Paxos. Highest ballot number based vote selection is provided by means of the routine `highest()`. Algorithm 3.30 shows the learning of the transaction status by the new leader within the context of *Phase* 1*b* of Paxos.

Algorithm 3.31 *leader.readAckVote(k, i, ballot, tmVote, wBallot)* Aggregates all acceptor votes, in terms of value *tmVote* and *wBallot*, for a item of key k and replication factor i. In case a majority of acceptor votes for the current leader *ballot* has been reached selected the vote of the highest ballot number *wBallot*. Finally, the selected vote is sent to all acceptors for acceptance.

```
1: ReadVotes[k][i][ballot] ∪ (tmVote, wBallot)
2: if |ReadVotes[k][i][ballot]| ≥ ⌊r/2⌋ + 1 then
3:     (hVote, hBallot) ← n.highest(ReadVotes[k][i][ballot])
4:     if hVote = ⊥ then
5:         vote ← ABORT
6:     end if
7:     for tm ∈ TMs do
8:         tm.vote(t_id, k, i, vote, ballot)
9:     end for
10: end if
```

The failure of a learner is handled in a similar fashion to the failure of an acceptor. However, the failure of the learner is detected by the fault detector and triggered with an suspicion message. To validate its failure suspicion the leader has to learn the status of the transaction with respect to the failed learner votes. Just like in the case of an acceptor failure, the leader has to start a new Paxos ballot with a higher ballot number and no actual value so as the learn the transaction status. In case no votes by the

suspected learner came through the leader will propose an $ABORT$ vote on its behalf eventually (Algorithm 3.31, line 4), thus causing the whole transaction to be aborted. Algorithm 3.32 shows the learner failure detection steps taken by the leader.

Algorithm 3.32 *tm.suspect(dtx, tp)* Creates a new ballot number for the given transaction identifier dtx. Starts a new Paxos ballot for the suspected learner tp for all of its affected dat items.

1: $IT \leftarrow \{(tp_i, i, item) | (tp_i, i, item) \in faultDetector.Data \wedge tp_i = tp\}$
2: **for** $(tp_i, i, item) \in IT$ **do**
3: $\quad n.propose(dtx, item.di.k, i, TMs, \perp, n.nextBallot(dtx))$
4: **end for**

Paxos commit with symmetric replication

The elaborated Paxos commit distributed transaction protocol can be applied to any distributed data management system so as to facilitate consistency of the managed data. We however, consider only Chord based DHT systems in conjunction with a symmetric replication scheme. To apply Paxos commit on top of a Chord system, so as to enhance symmetric replication for consistency, a number of Paxos commit algorithms have to be adapted to the features of the underlying data management system. Algorithms 3.33 – 3.37 address the access to nodes and data items of Paxos commit in a symmetric replication Chord. In case of a different data management system is used they have be adapted accordingly.

Regarding Paxos items that are subject to transactional processing, they do not convey key identifier space information. Since symmetric replication DHT data management operations are based on key identifiers Paxos items have to be mapped into KI. Thereby, the replica enumerator, provided by Paxos item counting (e.g., Algorithm 3.34), can be exploited for the *symm* operator so as to compute the equivalence class key identifier and to provide the replica enumerator.

To select the acceptors of a transaction (as in Algorithm 3.18, line 3) any node of the Chord DHT environment can be chosen. However, to inherently facilitate uniform load distribution among the DHT nodes, in terms of participation to acceptor sets, key identifier space mapping can be exploited. Precisely, by mapping the transaction identifier t_{id}, which can be generated randomly, into the Chord key identifier space, the consistent hash function will always uniformly distribute the transaction identifiers among all nodes of the environment. Consequently, these nodes will become the initial leaders of the corresponding transactions. To determine the remaining acceptor set managers the equivalence class of the mapped transaction identifier $dtx.id$ can be exploited. That is, all nodes responsible for an equivalence class containing the mapped transaction identifier $dtx.id$ can be assigned to the acceptor set as redundant transaction managers to the leader. Algorithm 3.33 shows the symmetric replication based retrieval of replication transactional managers, i.e., Paxos acceptors.

Similar to the acceptor set, the learner set retrieval (as in Algorithm 3.20, line 4) is bound to the underlying symmetric replication Chord DHT. This means that the learners have to be determined by mapping *item* (i.e., its key) into the key identifier space

Algorithm 3.33 *n.getTM(dtx, i)* Computes the equivalence class key identifier given the identifier of the transaction dtx and the replication factor enumerator i. The first successor node of the computed equivalence class key identifier is returned to be an acceptor set node.

1: $rep_i \leftarrow n.symm(f_h dtx.id, i)$
2: $tm_{succ} \leftarrow n.findSuccessor(rep_i)$
3: **return** tm_{succ}

and subsequently finding all responsible nodes for the key identifier space equivalence class of *item*. Algorithm 3.34 shows the retrieval of the learner set.

Algorithm 3.34 *n.getTP(item, i)* Computes the equivalence class key identifier given the *item* and the replication factor enumerator i. The first successor node of the computed equivalence class key identifier is returned to be a learner set node.

1: $rep_i \leftarrow n.symm(f_h(item.di.k), i)$
2: $tp_{succ} \leftarrow n.findSuccessor(rep_i)$
3: **return** tp_{succ}

To compute the keys that are used for locking of the data item (as in Algorithm 3.23, line 1) the equivalence class key identifiers are exploited. Algorithm 3.35 shows the symmetric replication based lock key computation.

Algorithm 3.35 *n.getRep(item, i)* Returns the equivalence class key identifier given the *item* and its replica enumerator i.

1: $rep_i \leftarrow n.symm(f_h(item.di.k), i)$
2: **return** rep_i

Retrieval of data items from the local symmetric replication DHT storage (as in Algorithm 3.23, line 2) is shown in Algorithm 3.34.

Storage of data items to the local symmetric replication DHT storage (as in Algorithm 3.28, line 7) is shown in Algorithm 3.37.

During the course of a Paxos commit transaction nodes that do not participate to a transaction itself can join and leave the execution environment. In the context of the applied symmetric replication DHT this means that node responsibilities over data items are subject to change. In case a data item that is participating to the transaction changes its responsible DHT node, the new responsible DHT node has to be informed about the outcome of the transaction so as to stay consistent. To update the DHT nodes with the latest values of the newly assigned data items the corresponding learner of the transaction has to propagate the outcome to the DHT node. In turn, newly assigned nodes have to subscribe at the learners for data items of interest that are involved in ongoing transactions. Algorithms 3.38 – 3.40 address the address the propagation of transaction outcomes to newly assigned responsible nodes in the context of a symmetric replication Chord.

To adapt for Paxos commit a symmetric replication DHT node has to feature learner subscription capabilities for newly assigned data items. Whenever a DHT node extends

Algorithm 3.36 *n.getItem(item, i)* Computes the equivalence class storage key of *item* with respect to the replication factor enumerator i. Uses the storage key and i so as to retrieve the item at the local node n.

1: $ki \leftarrow n.getRep(item, i)$
2: **return** $di \leftarrow n.get_s(ki, i)$

Algorithm 3.37 *n.putItem(item, i)* Computes the storage equivalence class storage key of *item* with respect to the replication factor enumerator i. Uses the storage key to store item at the local node n.

1: $ki \leftarrow n.getRep(item, i)$
2: $n.put_s(item.di, ki, i))$

its key identifier space partition it has to query the replicas sharing the same equivalence class so as to restore replication factor of data items. In a Paxos commit based symmetric replication, the extension of the key identifier space partition, induced by the change of the predecessor, all equivalence class sharing replicas have to be queried. Only when a majority of equivalence class sharing replicas features the same data item value, consistency of the value is guaranteed, and it can be taken over by the newly assigned DHT node. Algorithm 3.38 shows the adaptation of the symmetric replication `updatePredecessor()` routine to Paxos commit.

Algorithm 3.38 *n.updatePredecessor(n')* applies the *symm* on the extended key identifier space partition interval determined by the new predecessor n' and sends queries the all replica nodes of the same equivalence class, by means of $obtainInterval()$.

1: $end \leftarrow f_n(predecessor)$
2: $start \leftarrow f_n(n')$
3: $predecessor \leftarrow n'$
4: **if** $start < end$ **then**
5: **for** $i \in R \setminus 1$ **do**
6: $x \leftarrow n.symm(start, i)$
7: $y \leftarrow n.symm(end, i)$
8: $n_{rsp} \leftarrow n.findSucc(x)$
9: $n_{rsp}.obtainInterval(x, y, n)$
10: **end for**
11: **end if**

Whenever a DHT node is requested to yield data items of a key identifier space interval, by means of an *obtainInterval* request, it has to examine the data items for involvement in Paxos commit transactions. In case they are involved in transactions they are not propagated, rather the requesting node is subscribed for the outcome of the transaction upon which the propagation will be made. To combine them, a symmetric replication DHT has to share the *TxItems* and *Subscribers* data structures with Paxos commit. Algorithm 3.39 shows the adaptation of the symmetric replication `obtainInterval()` routine to Paxos commit.

Algorithm 3.39 *n.obtainInterval(x, y, n′)* given the bounds x and y selects all data items located within these bounds at node n. It returns the data items to the specified node n' by means of the *replicate*() operator if they are not participating to transactions. Otherwise, subscribes n' to the outcome of the involved transactions.

```
1: DataItems[][] ← ∅
2: for i = x to y do
3:     for j =∈ R do
4:         dataItem ← n.get_s(i, j)
5:         InTx ← {(tx, k)|∀(tx, k) ∈ TxItems ∧ k = dataItem.di.k})
6:         if InTx = ∅ then
7:             DataItems[i][j] ← dataItem
8:         else
9:             for (tx, k) ∈ InTx do
10:                Subscribers[dataItem.di.k] ∪ (n′, tx)
11:            end for
12:        end if
13:    end for
14: end for
15: n′.replicate(x, y, DataItems)
```

All received data items at the newly responsible DHT node have to be buffered first till a majority of replica nodes responses have arrived. Only when a majority of replicas share the same data item value and this value features a higher timestamp then the currently stored one (if any), global serializability is preserved and the received data item can be accepted.

Propagation of data item transaction outcomes does not only restores the replication factor but also it preserves consistency, i.e., global serializability. Omitted updating of the data items will result in inconsistency among the replicas and incapability of future transactions to change the data item. Recap, an optimistic serializability locking attempt in face of outdated replica version (even at one node only) will never result in a *PREPARED* vote and thus no commit.

Paxos commit discussion

Although Paxos commit facilitates a predictable (i.e., F failures out of $2F + 1$ acceptors) reliability and consistency of distributed transactions, the enhanced reliability does not come without price. In general, Paxos commit introduces a significant message overhead as compared to 2PC and additional message flow delays.

Whereas 2PC features a linear complexity $O(r)$, in terms of exchanged messages among participants, with respect to the replication factor r, Paxos commit features a quadratic complexity $O(r^2)$. Namely, 2PC requires in an ideal case, which is free of conflicts and failures, for each phase r messages. Given the four phases of 2PC (i.e., *prepare*, *prepareAck*, *commit* and *commitAck*) $4r$ messages are necessary to complete a transaction.

Algorithm 3.40 *n.replicate(x, y, DataItems)* stores to the buffer *TempData* at node n all received *DataItems* after rearranging them so as to match the local equivalence class key identifiers and replica identifiers. Once the buffer features a majority of data items it stores value locally only if it is the latest one.

```
1: for i = x to y do
2:     it ← n.symm(i, r)
3:     for j ∈ R do
4:         dataItem ← DataItems[i], [j]
5:         rep ← n.symm(j, r)
6:         TempData[(it, rep)] ∪ dataItem
7:         if |TempData[(it, rep)]| ≥ ⌊r/2⌋ + 1 then
8:             item_l ← n.latest(TempData[(it, rep)])
9:             item_c ← n.get_s(it, rep)
10:            if item_l.ts > item_c.ts then
11:                n.put_s(item, it, rep)
12:            end if
13:        end if
14:    end for
15: end for
```

On the other hand, Paxos commit features 7 phases (i.e., *initPaxos*, *registerPaxos*, *setTM*, *prepare*, *vote*, *voteAck* and *decision/outcome*) out of which two phases feature more than just r messages. That is, the *decision/outcome* phase features $2r$ messages as r messages have to be sent to the acceptors and r to the learners. In turn, the *vote* phase features r^2 messages as each of the r learners has to inform the r acceptors about its status. Hence, due to all phases and their individual message flows an ideal Paxos commit amounts to $r^2 + 7r$ messages in total. Given the quadratic complexity (i.e.,$O(r^2)$) Paxos commit is generally considered to be prohibitively expensive for replication factors higher than 5 (i.e., $r > 5$).

The aforementioned analysis of Paxos commit and 2PC assumes only one data item subject to the transaction. In case of more data items, the complexities increase even more and can be multiplied by the number of items i, thus making the protocols even more expensive, i.e., $i(r^2 + 7r)$.

4

Distributed Workflow Management Model

While the previous chapter introduced concepts w.r.t. the distributed management of data, in this chapter theoretical background of our work is provided that addresses the management of workflow definitions in a distributed fashion. This includes the formal workflow definition model, distributed execution control concepts and systems descriptions that conduct them. The complete model covers even the service heterogeneity aspects of the use case scenario (i.e., modern emergency management) as discussed in Chapter 2. That is, it compasses discrete and continuous service types as well. Towards the end of this chapter an example of the distributed execution model is provided so as to illustrate their applications in similar setting as in the motivation use case scenario. The concepts elaborated in this chapter lay the formal groundwork for our contribution in the upcoming chapters, but to some extent also correspond to the state-of-the-art in the domain of distributed workflow execution.

4.1 Workflow Management

In order to comprehensively understand workflow definitions, we provide in this section a formal model that declares them. Precisely, a workflow definition specifies a dynamic execution process that combines various entities of a distributed system so as to accomplish a comprehensive functionality or goal. The cornerstone of the workflow definition are the services which represent steps in the execution process. That is, a service and its corresponding invocation represents one step in the process of a workflow definition execution.

A service encompasses a set of atomic operations that are subject to completion upon its invocation. These operations are usually hidden from the end-user in a black-box fashion. Different internal configurations of atomic operations constitute different types of services. Different service types can interact among each other by exchanging data that is produces as a consequence of their executions.

In order for a service type to be compatible with any other service types, in terms of data exchange, all service types have to adhere to a common data space. The common data space can be encoded by means of a common data alphabet Φ. This way data can be communicated among the service types and consumed without any preconditions. One unit of communication among service types is the *data item – $di \in DI$* that conveys plain payload information, represented with the common data alphabet Φ, and is identified by unique key.

The payload conveyed by a data item can correspond to any entity (e.g., a blob, a String, a complex data structure etc.) that can be encoded by means Φ. Data items may also convey an empty information payload. Such data items are featured by service types that are not relying on data at all (e.g., delay activities) for activation. As suggested in the motivation chapter (i.e., Section 2.3) there might be different kinds of interaction among the services when it comes to transfer data among them. While some services feature simple transmissions of a few data items, others feature continuous transmissions of potentially unlimited data item numbers. At the same time, the setting of the execution environment does not have to be limited to one type of service only as suggested in Section 2.3. Various (i.e., hybrid) combinations of services of different interaction scenarios should be possible as well. To encompass the different interaction scenarios among with our model we distinguish between *discrete* and *continuous* service types.

In general, discrete service types are characterized by short lasting execution duration and limited produced/consumed data volumes. Moreover, discrete services do not exhibit any internal state. In case they do, it is generally hidden away from the workflow definition (i.e., the end-user) in a black-box fashion. Activation of discrete service types is performed at runtime by means of invocation requests, that are usually accompanied by data. Thereby, the invocation data features a finite set of semantically unrelated data items. Once activated, the discrete service type executes its internal operations in processing the incoming data items and produces new ones as the invocation result. The newly produced data items can subsequently be transferred to other service types which react to them in the same way. The actual transformation of incoming data items into outgoing data items is determined by the service type (possibly stateless) business logic that is hidden away from the end-user in a black-box fashion. Once its execution finishes, the discrete service type is not any more active from a workflow definition execution point of view. Moreover, the finished service type is free to process new requests from other workflow definitions in a completely new context independent of the previous one. This style of service type invocation constitutes the request-response fashioned discrete execution of data. Formally, the discrete service type definition is provided as follows:

Definition 4.1 (Discrete service type – s_d)**.** *A discrete service type s_d is defined as tuple $s_d = (id, I_d, O_d, f_p)$*

- *$id \in \Phi$ is the service type identifier,*
- *I_d is the set of input data items that are subject to processing with:*

$$I_d \subseteq DI^n \textbf{ with } n \in I\!N^+ \cup \{0\}$$

- *O_d is the set of output data items that have been produced as a result of processing with:*

$$O_d \subseteq DI^m \textbf{\textit{ with }} m \in I\!N^+ \cup \{0\}$$

- *f_p is the black-box business logic function that processes input data items into output data items:*

$$f_p : I_d \mapsto O_d$$

□

Note, that a discrete service type can feature multiple input data items stemming from different service types that are needed for its activation (i.e., $n > 1$). These are referred to as the joining discrete service types. Likewise, multiple data items can be created as output (i.e., $m > 1$) that are delivered to different service types for activation, as well. These kind of service types are referred to as the forking discrete service types. In general, the input and output data items cardinalities do not have to match necessarily (i.e. $n \neq m$) and are dependent on the internal business logic.

However, not all service types tend to finish their execution in a foreseeable amount of time. For instance, sensor device hosted service types execute their operations indefinitely once activated. As a consequence of indefinite activation, continuous service types tend to produce massive amounts of data items that are subject to consumption by other service types of interest.

The processing of continuous data cannot be operated randomly on a first-come, first-serve basis. This way the data production semantics of the continuous service type will not be preserved in face of inherent distributed systems characteristics such as network congestions. Rather, a continuous flow of data is always subject to a logical total order that reflects the creation time of the containing data items. In other words, a continuous flow data item that has been produced before another data item must also be sequentially processed before that other data item at a consuming service type. The totally ordered continuous flow of data items is referred to as a *data stream* and is formally defined as follows:

Definition 4.2 (Data stream – DST)**.** *A data stream DST is defined as potentially infinite and totally ordered set of tuples $DST = \{(i, di) | i \in I\!N^+, di \in DI\}$ where i corresponds to the logical creation sequence number within the data stream and di to the data payload information represented by means of a data item.* □

Since no service hosting node is physically capable of locally managing a potentially infinite stream of data, the results of data stream processing have to be stored in some form at the node. To this end service types employing the continuous data stream introduce a internal state into which transient streaming data items are condensed (i.e., transformed) into. This internal state is subject to constant change and reflects the precessing results on the whole data stream encountered so far. However, since the state affecting data streams are potentially infinite so can the state changes be. To denote the continuously changing internal state of all data stream employing service types the infinite set SS is used:

$$SS = \{ss_1, ss_2, ss_3, ...\}$$

The actual change in state is determined by the service type business logic (i.e., its internal operations) that is hidden away from the end-user in a black-box fashion. The business logic specifies how input data items are used to change the internal state and how this change is propagated downstream by means of output data items. Once processed and reflected in the internal state, each input stream data item can be physically discarded at some point in time, thus offering room for future data item processing. The businesses logic however, does not have to rely on input data necessarily so as to change the internal state. When subjected to temporally scheduled processing business logic may alter the internal state even though no data items are available at the input stream. Formally, the continuous service type is defined as follows:

Definition 4.3 (Continuous service type – s_c)**.** *A continuous service type s_c is defined as tuple $s_c = (id, ss, I_c, O_c, f_c)$ where:*

- *$id \in \Phi$ is the service type identifier,*
- *$ss \in SS$ is the current internal state,*
- *I_c is the potentially infinite set of data items as a Cartesian product over i input data streams with:*

$$I_c = \prod_{x=0}^{i} DST_x \textbf{ with } i \in IN^+$$

- *O_c is the potentially infinite set of data items as a Cartesian product over o output data streams with:*

$$O_c = \prod_{x=0}^{o} DST_x \textbf{ with } o \in IN^+$$

- *f_c is the black-box business logic function that processes input streaming data items in conjunction to the state to a new service type state and output data items:*

$$f_c : I_c \times SS \mapsto SS \times O_c$$

□

Note, that a streaming service types can feature multiple input (i.e., $i \geq 0$) and output (i.e., $o \geq 0$) streams whose cardinalities do not have to match necessarily (i.e., $i \neq o$). These kind of continuous service types are responsible for the merging and/or forking of the data flow. While forking a data stream is as simple as sending the same output data item of a the stream to multiple disjoint service instances, merging is not as trivial. Usually, windows on the incoming data streams, in terms on the number of data items, are asserted which are then joined based on the specific implementation of the service type business logic. Since data stream joins are streaming service type implementation specific, they are not further discussed in the follow up of this section. Other important information about streaming service types such as currently processed data items for each input/output data streams are subsumed inside the internal state $ss \in SS$.

Observe from the previous definition that continuous service types do not have to feature any incoming or outgoing data streams (i.e., $i = 0 \vee j = 0$) as well. In order to qualify as of continuous type service types should not lack both input and output streams at the same time. Continuous service types that do not feature input data streams but feature output data streams (i.e., $i = 0 \wedge j \neq 0$) are referred to as *data sources*. Such service types are usually located at sensor devices and continuously produce data items and for the subsequent service types to consume. Continuous service types that do not feature output data streams but feature input data streams (i.e., $i \neq 0 \wedge j = 0$) are referred to as *data sinks*. Such service types are the end consumers of the data streaming stemming from the data sources.

Given all possible discrete and continuous service types they can be encompassed by one general set of all service types. The formal definition of the general service type set is provided as follows:

Definition 4.4 (Service types – S)**.** *Let S_d be the set of all discrete service types and S_c be the set of all continuous service types. The set of all service types is defined as the union of the discrete and continuous service type sets:*

$$S = S_d \cup S_c$$

□

When it comes to deployment, all service types, both continuous and discrete, can be offered by numerous nodes of an execution environment. The actual deployment of a service type at a node corresponds to an instance of that service type. A workflow definition execution environment is thus comprised by the sum of all service instances. The formal definition of a service instance is provided as follows:

Definition 4.5 (Service instances – SI)**.** *The set of all service instances SI is defined by the binary relation $SI \subseteq (S \times N)$ where S is the service type set and N is the set of nodes.* □

Service types and their concrete instances are the cornerstones of workflow modeling. They are subject to combining for execution and data exchange so as to create comprehensive high-level functionality. Finally, the continuous data item model, of this thesis corresponds to an abbreviated version of the formal data stream management model as introduced by the OSIRIS-SE [BS11a] framework.

4.1.1 Workflow Definition Structure

A workflow definition is constituted by a set of service types and their binary relations. The relations among the service types are structurally ordered so as to uphold the precedence semantics of the workflow execution process. That is, an order of service types invocations has to be specified that ultimately leads to the comprehensive functionality of the end-user application modeled by the workflow definition.

In order to uphold the structural semantics of a workflow definition, in terms of invocation precedence orders, instances of the specified service types have to be traversed and invoked. Service instances that have been invoked and are currently active (i.e., in

case there are concurrent ones) participate to the execution workflow definition and share their results with the subsequent service instances. The consecutive invocation of service instances in a ordered fashion is referred to as the *control flow*.

The execution of workflow definitions is additionally characterized by the data that is exchanged among the service instances at each step of the execution. At service instance invocation time data is consumed, and as a result of it new data is produced that is in turn forwarded to the subsequent service instance for consumption and so on.

However, the nature of the encompassed activities does not have necessarily to coincide the with the nature of the control flow model from a data exchange point of view. In practice, there might exist activities that are not relying on data at all (e.g., a delay or synchronization activities) for invocation. On the other hand, there might exist activities that are relying on data that has been produced far back in the precedence order of the control flow. Data agnostic service types can be omitted from the structure of the workflow definition in the context of data exchange but not in the context of the control flow.

Since the control flow cannot accurately capture all possible scenarios of data exchange a new model has to be introduced. The new model orders data exchange so as to conform with the flow of control. For instance, only invoked activities can produce data and exchange this data with other activities that are also subject to invocation. The control flow bounded and ordered exchange of data constitutes the *data flow* of the workflow definition. For the data flow to coincides with the control flow its data exchange relations have to correspond to a sequence of service invocation control flow relations. This implies the relations of data flow model to be transitive closures of the control flow relations. With respect to the structure of the control flow and data flow relations a formal workflow definition is described as follows:

Definition 4.6 (Workflow definition – wf)**.** *A workflow definition wf is defined as tuple $wf = (A, \prec_{cf}, \prec_{df})$ where:*

- *A is a finite set of activities that corresponds to a subset of service types that are subject to invocation with:*
$$A \subseteq S$$
- *$\prec_{cf}$ is the finite set of control flow binary relations among the activities, such that they represent the invocation order of the specified activities with:*
$$\prec_{cf} \subseteq (A \times A)$$
- *$\prec_{df}$ is the finite set of data flow binary relations among the activities such that they represent the data exchange order of the specified activities and they are transitive closures of the control flow relations with:*
$$\prec_{df} \subseteq (A \times A)$$

The set of all workflow definitions is denoted as WF. □

Note, that activities correspond to invocations of service types. The actual selection of the concrete service instance to carry out the activity takes place at runtime. This

is referred to as *late binding* of the activity to a service instance. Regarding the control flow, the precedence order semantics represent the order in which the service types are invoked. For instance, a relation $(x, y) \in \prec_{cf}$ with $x \in A$ and $y \in A$ refers to the fact that an instance of service type x is invoked before an instance of service type y. On the other hand, in the context of the data flow, the precedence order semantics represent the order in which data is produced and subsequently consumed. For instance, a relation $(x, y) \in \prec_{df}$ with $x \in A$ and $y \in A$ refers to the fact that activity y consumes data which is produced by activity x.

In order to ensure semantically meaningful structuring of workflow definitions which result in coherent wholes, we mandate the workflow definition structures to be strongly connected. Strong connectivity of workflow definition structures implies control flow precedence order relations that guarantee invocations of all specified activities inside the workflow definition. Activities that will not be eventually invoked, given thee control flow relations structure, cannot exist in a strongly connected workflow definition.

Strong connectivity necessitates our service type model to be further enhanced with two artificial service types which denote the *start* and the *end* of a workflow definition execution. In other words, only the invocation of those two special service types can start/end the workflow definition execution process, respectively. While structuring a workflow definition, we restrict ourselves to only one encompassed start activity, whereas the end activity can be encompassed multiple times. To achieve strong connectivity there has to exist a transitive closure inside the control flow between the start activity and the set of end activities. Moreover, any activity has to be in a transitive closure relation with the start activity and the set of end activities w.r.t. the control flow. For all the aforementioned structural requirements we specify a well-formed workflow definition formally as follows:

Definition 4.7 (Well-formed workflow definitions)**.** *A workflow definition wf is well-formed if:*

- *For each workflow definition there exist only one start activity a_s and a set of end activities A_e with:*
$$\{a_s\} \in A, A_e \subseteq A$$
- *With respect to the control flow there is no other activity that comes before the start activity a_s and no activity that comes after the end activities in A_e:*
$$\forall a_x \in A \setminus a_s : \nexists (a_x, a_s) \in \prec_{cf} \text{ and } \forall a_y \in A \setminus A_e \wedge \forall a_e \in A_e : \nexists (a_e, a_y) \in \prec_{cf}$$
- *Every activity of A_e is transitively connected to a_s in $\prec_{cf}$,*
- *Every activity $a_x \in A \setminus \{a_s \cup A_e\}$ is transitively connected to a_s and at least one activity of A_e in $\prec_{cf}$.* □

By being strongly connected from an control flow point of view, workflow definitions are also strongly connected from an data flow point of view. Recap, all data flow activity relations are transitive closures of strongly connected control flow relations,

hence they also must be strongly connected. This implies that for any data producing activity, except the artificial start and end activities, there must be an activity that consumes this data.

Analogous to SOA encapsulation whole workflow definitions themselves can map to superordinate service types in which case they can be used recursively as activities of other workflow definitions. The structure of a workflow definition can be represented with a directed graph that consists of nodes and edges among them. Naturally, the nodes correspond to the activities, whereas the edges to precedence order relations among them. Given the two types of relations among activities (i.e., data and control flow) there also exist two types of graphs.

Example 4.1

Figure 4.1 illustrates an example of a well-formed workflow definition. The example workflow corresponds to a simplified version of the workflow as introduced in the motivation chapter, i.e., Figure 2.3. As this figure shows this structure is composed of six activities. Namely, the *start* activity, the *location* activity, the *weather* activity, the *map* activity, the *heat-map* activity and the *end* activity. The precedence order relations of the control flow are depicted with the directed arrows, such that the preceding activity directs the arrow to the succeeding activity. In accordance Definition 4.7 the start and end activities are necessary so as to start and end the execution.

This workflow definition is structured in such a way so that start activity precedes the location activity. This service type queries the GPS location of an person given its id with the start service type. Once the coordinates of this person have been determined they are used to query the weather as well as the topological map of these coordinates by means of their corresponding activities. The querying of the weather activity and the map activity can be achieved in parallel as they are independent activity. Once all of the data has been retrieved it is sent to the heat-map activity which subsequently computes a heat-map for the specified person. As the heat-map activity precedes the end activity the execution of the workflow definition is finished by sending the heat-map to the end activity.

4.1.2 Workflow Definition Execution

In order to uphold the structural semantics of a workflow definition regarding the precedence order of invocations, the specified activities have to be traversed and invoked in an ordered fashion at runtime. This implies certain service instances only to be invokable once the invocation of their preceding service instances has finished and not before that. Given the characteristics of a distributed system such as concurrency, fault-tolerance and scalability of service instances, one can assume that no two executions of a workflow definition are likely to behave in the exact same way.

To describe the runtime execution behavior of a workflow definition late-binding of service instances is considered. Precisely, at runtime a workflow definition execution exhibits a dynamically changing state that features the currently invoked (i.e., late-bound)

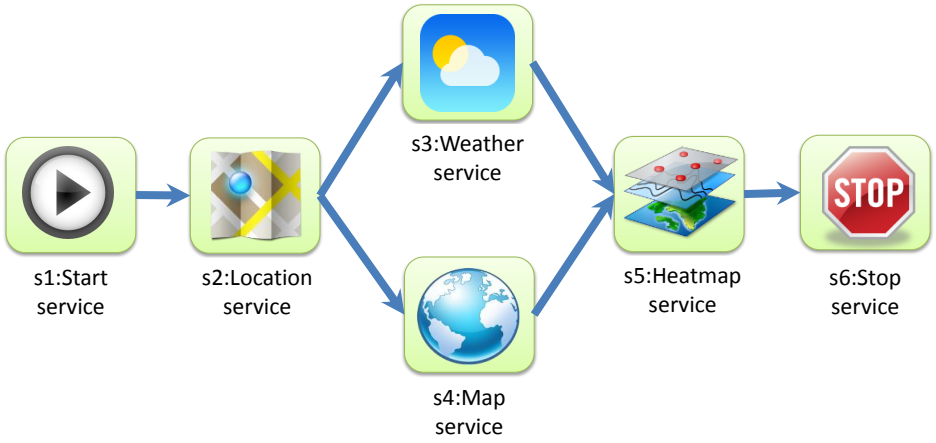


Figure 4.1: The structure of a workflow definition.

activities, the competed activities and the subsequent activities that still have to be late-bound. Moreover, the workflow execution state features the set of service instances that are currently late-bound of the invoked activities. To keep a track of the late-binding in the course of the workflow instance execution a history of the invoked service instances is maintained as well. The dynamically changing execution state is conveyed by means of an instance of the workflow definition that traverses the specified service instances and triggers their execution. Formally, an instance of a workflow definition is defined as follows:

Definition 4.8 (Workflow instance – *wi*)**.** *An instance wi of a workflow definition is defined as tuple* $wi = (wf, A_c, A_a, A_s, SI_c, SI_a)$ *where:*

- $id \in K$ *is the unique identifier of the workflow instance,*
- $wf \in WF$ *is the workflow instance's definition,*
- A_c *is the subset of activities of the workflow definition* wf *that have been completed with* $A_c \subseteq wf.A$
- A_a *is the subset of activities of the workflow definition* wf *that are currently active with* $A_a \subseteq wf.A$
- A_s *is the subset of activities of the workflow definition* wf *that are subsequent for activation invocation with* $A_a \subseteq wf.A$
- SI_c *is the finite subset of service instances that have carried out the execution of the completed activities with:*

$$SI_c \subseteq SI \textbf{ where } \forall a_c \in A_c \exists si_c \in SI_c : si_c.s = a_c$$

- SI_a *is the finite subset of service instances that are carrying out the execution of the currently invoked activities with:*

$$SI_a \subseteq SI \textbf{ where } \forall a_a \in A_a \exists si_a \in SI_a : si_a.s = a_a$$

The set of all workflow instances is denoted as WI. □

Note, that the subsets are A_c, A_a and A_s are all disjoint but together correspond to the activity set of the workflow definition wf:

$$A_c \cap A_a \cap A_s = \emptyset \ \wedge \ A_c \cup A_a \cup A_s = wf.A$$

In practice, the existence of multiple instances of the same workflow definition is very likely, e.g., in case of parallel branches. Our model mandates however that each instance can only be in charge of its own state and does not interfere with the state of other instances. Multiple instances do not share their execution state for the purpose of collaborative change. From an practical point of view however, this assumption strongly depends on implementation of the storage facility that manages the instances and might be capable of accessing other instances.

The state of the workflow instance is subject to change in the course of time. During the execution of a workflow instance activities are invoked and subsequently completed only to be succeeded with new activities of the predefined control flow. The change to the invoked activity set A_a in time is consequently attended to in the active service instances set SI_a and the completed service instance set SI_c. That is, active service instances that carry out the execution of activities in A_a will be located in SI_a whereas the finished ones in SI_c. The workflow instance state does not have to change only w.r.t. the activity sets. Activities of longer lifespan are also subject to change in time w.r.t. their internal state. For example, streaming service instances continuously process incoming/outgoing data stream objects and as a result change their internal state. The temporal change to workflow instance execution state is summarized as follows:

Corollary 4.9 (Temporal workflow instance state transition)**.** *The temporal version history of all workflow instances is defined by the binary relation $WI_t \subseteq (T \times WI)$ where T is the timestamp set and WI workflow instance set subject to versioning.*

A workflow definition instance wi has advanced its execution state if in the instance history WI_t exists a newer version of the instance:

$$\exists (t_x, wi), (t_y, wi) \in WI_t \implies t_x < t_y$$

□

Given that service instances can have different state change semantics, we have to distinguish the temporal changes of workflow instance by taking separately into consideration the control-flow and the data-flow. Any change to workflow instance state, induced by the flow of control, necessary implies the change in one of the activity sets, i.e., A_c, A_a or A_s. Control flow based invocations and completions of service instances cause transitions of the corresponding activities through the activity sets starting at A_s and ending at A_c. This is in particular the case for service instances of discrete type. Changes to workflow instance state, induced by the flow of data, implies the change to the internal state of the activated service instances. The data flow caused transition of data items among the activated service instance results in data item transformation and

possibly change of the internal state of the traversed service instances. Data flow based change to workflow instance state mainly affects streaming service instances.

The control flow based change of a workflow instance w.r.t. its activity sets (i.e., A_c, A_a, A_s) between two versions $(t_x, wi), (t_y, wi) \in WI_t$ is denoted as:

$$wi.A_c \xrightarrow{\prec_{cf}} wi.A'_c$$

$$wi.A_a \xrightarrow{\prec_{cf}} wi.A'_a$$

$$wi.A_s \xrightarrow{\prec_{cf}} wi.A'_s$$

This implies that the set of currently active activities A_a of the workflow instance wi at timestamp t_x has changed at timestamp t_y because:

- Either some activities have finished their execution and have flowed over into the completed activity set:

$$wi.A_c \xrightarrow{\prec_{cf}} wi.A'_c \implies A_c \neq A'_c \wedge (wi.A'_c \cap wi.A_a) \neq \emptyset \wedge (wi.A_c \cap wi.A'_a) = \emptyset$$

- Or some activities scheduled for activation have been activated and have flowed over into the currently active set:

$$wi.A_s \xrightarrow{\prec_{cf}} wi.A'_s \implies A_s \neq A'_s \wedge (wi.A'_s \cap wi.A_a) = \emptyset \wedge (wi.A_s \cap wi.A'_a) \neq \emptyset$$

If at least one of the two cases occurs and either A_s or A_c over time then A_a must also change for wi by default:

$$wi.A_a \xrightarrow{\prec_{cf}} wi.A'_a \implies wi.A_c \xrightarrow{\prec_{cf}} wi.A'_c \vee wi.A_s \xrightarrow{\prec_{cf}} wi.A'_s$$

Naturally, the changes to the invoked activity set A_a have to be accompanied by changes of service instance sets SI_a and SI_c. For every invoked activity there has to exist a corresponding service instance in SI_a that carries out the execution. Likewise, activities that have been completed and are not any more located inside the active set A_a should represented with service instances in completed service instance set SI_c:

$$wi.SI_a \xrightarrow{\prec_{cf}} wi.SI'_a \implies (wi.SI_a \setminus wi.SI'_a) \cap wi.SI'_c \neq \emptyset$$

Given the temporal changes of a workflow instances, at runtime activities of a workflow definition are flowing over from the subsequent activity set A_s to completed activity set A_c through the invoked activity set A_a. Hence, the initial state of a workflow instance wi assumes only the subsequent activity set A_s to feature activities, whereas the other ones to be empty:

$$\exists (t_s, wi) \in WI_t \wedge \nexists (t_x, wi) \in WI_t :$$

$$t_x < t_s \wedge wi.A_c = \emptyset \wedge wi.A_a = \emptyset \wedge wi.A_s \neq \emptyset \wedge wi.SI_a = \emptyset \wedge wi.SI_c = \emptyset$$

The end state of a workflow instance is not as clear as the initial. In the presence of non-deterministic activities in the workflow definition, some activity branches might not get invoked at runtime. As a consequence, they will never leave the subsequent activity set A_s and feature corresponding service instances in SI_c. This implies that A_s does not have to be completely empty for a workflow instance to complete its execution. Hence, we conclude a workflow instance to be completed if it does not feature active service instances and if there does not exist a newer one in the temporal version history WI_t:

$$\exists(t_e, wi) \in WI_t \wedge \nexists(t_x, wi) \in WI_t :$$

$$t_x > t_e \wedge wi.A_c \neq \emptyset \wedge wi.A_a = \emptyset \wedge wi.SI_a = \emptyset \wedge wi.SI_c \neq \emptyset$$

Advanced Discrete Workflow Execution Models

The distributed workflow execution model naturally assumes well-formed structures of workflows so as to guarantee successful completion of a distributed execution. Given the inherent characteristics of distributed systems such as node failures or uniqueness of certain service types, the distributed model is subject to advancement so as to guarantee the termination of a workflow instance at runtime for any kind of unforeseeable circumstances. For example, if a service type that is provided at only one node fails as a consequence other workflow instances in the context of their subsequent activity sets are affected. Since invocations of encompassed activities of a workflow definition are of atomic nature it is desirable to extend atomicity of execution on the whole workflow definition, as well. In turn, the whole workflow definition can be abstracted as an atomic service type itself and encompassed by other workflow definitions.

By enhancing the distributed workflow execution model along the lines of activity semantics, alternative execution paths of the same semantical meaning, can be supported so as to guarantee termination of execution for any kind of activity outage. Semantic distributed workflow execution implies the distinction of all activities into *compensatable, retriable* and *pivot* activities in conjunction to a preference order control flow.

- Compensatable – A compensatable activity features an inverse that can undo its effects on the execution. The compensatable-inverse activity pairs have to be explicitly specified.
- Retriable – A retriable activity features multiple retires until its invocation is successful.
- Pivot – A pivot activity is neither compensatable nor retriable. Pivots are the safepoints of the semantic execution.

While the semantically alternative execution paths need to be constructed out of the aforementioned activity types their sequential ordering is explicitly determined by means of the preference order. Precisely, the preference order defines paths that lead back to the most recent pivots, by activating compensating activities, upon which an alternative (composed of retriable activities) path may be chosen. If provided explicitly at the time of workflow structuring, precedence order paths guarantee termination of

execution at runtime. By falling back to alternative paths in case of node failures the execution of workflow instances can be always continued.

In order to adhere to the advanced distributed workflow execution models, preference order paths need to be added to the workflow definition model explicitly. This in turn necessitates all compensation activities to be added to the subsequent activity set A_s at all workflow instances whenever a node failure occurs. The compensation activities can be added dynamically for each workflow instance, while recovering node failures, or at workflow instance creation time. The later case however, implies that workflow instances can finish their execution with an non empty subsequent activity set (i.e., $A_s \neq \emptyset$ – in case there are no failures). Regardless of the approach, addition of new activities to the subsequent activity set A_s enables atomicity of the distributed workflow instance execution model.

Streaming Workflow Execution Model

Execution of workflow instances which are encompassing streaming service types are not as much influenced by the control flow as in the case of discrete service types. At best, the control-flow represents for the streaming instances the order in which they are invoked (i.e., activated). Rather streaming instances of a workflow feature a consequent sequence of A_s set changes upon which they are all continue to be active for a longer period in time. During this time the streaming service instances jointly process the transient data streams and will not change from a control flow point of view any more. At a much later point in time, once all the data streams have been processed streaming workflow instances feature a consequent sequence of A_c set changes which effectively stop their execution.

Therefore, streaming service instances are predominantly influenced by the data flow instead of the control flow. Caused by the continuous processing of data streams for any given point in time streaming service instances will feature an updated internal state (i.e., $si.s.ss$) of the corresponding service type. Either they will have processed an input data item and as a consequence of this updated the internal state, or they will have produced an output data item based on the current state.

Hence, the state of a workflow instance, that is characterized by a continuous data flow, can at a certain point in time (i.e., wi_t) be represented with the state function f_{st} as the Cartesian product over the internal states of each active streaming instance (i.e., $wi_t.wi.SI_a$) as follows:

$$f_{st} : WI_t \mapsto SS^i \textbf{ with } i \in I\!N^+ \cup 0 \textbf{ where } f_{st}(wi_t) = \prod_{si \in wi_t.wi.SI_a} si.s.ss$$

The data flow based change of a workflow instance wi w.r.t. its active service instance set SI_a between two versions $(t_x, wi), (t_y, wi) \in WI_t$ is denoted as:

$$wi.SI_a \xrightarrow{\prec_{df}} wi.SI'_a$$

This temporal change implies that there is at least one streaming service instance has changed its internal service type state and two temporal versions of the same instance cannot be same w.r.t. to f_{st} :

$$wi.SI_a \xrightarrow{\prec_{df}} wi.SI'_a \implies wi.SI_a = wi.SI'_a \textbf{ where}$$

$$\forall (t_x, wi) \in WI_t \nexists (t_y, wi) \in WI_t : f_{st}((t_x, wi)) = f_{st}((t_y, wi))$$

In general, there is more to the internal state of a continuous service type than just plain data item processing results. Internal state conveys also other important runtime information such as unprocessed data items of the corresponding input/output data streams. Input stream data items usually reside (sequentially ordered) inside buffers and wait for their turn to be processed by the business logic. Likewise, output stream data items also reside in buffers and wait to be sent through the network to the subsequent service instances. While waiting for their processing turn streaming data items are subsumed by the internal state of a service instance. More details on streaming services state in conjunction to a formalized description of it can be found under [BS11b].

The continuous data flow can be in a real world distributed system subject to interruption due to network congestions and failures etc. Especially if mobile nodes are involved in workflow instance, data items of a stream might get lost or delayed, nodes processing them might fail etc. For most end-user application scenarios even the slightest disruption of the data flow (e.g., the loss of a single streaming data item) is not an option as it might cause the loss of crucial information (e.g., the emergency management scenario). In order to prevent the disruption of the continuous data flow due to node failures, the internal state of activated service instance (and everything it subsumes) has to be applied redundantly. To this end streaming systems generally apply one of the redundancy strategies, that are widely known in literature as i.e., *active-standby* and *passive-standby*.

Active-standby imposes the creation of completely redundant and secondary data flow that is carried out by redundant service instances in parallel. This redundant data flow is only used in case there is a failure in the primary data flow, otherwise the transient data items are discarded. Given a highly voluminous data flow, that is composed of numerous service instances, active-standby is commonly considered as a very expensive redundancy strategy w.r.t. resource consumption. A far more resource friendly approach to data flow redundancy, i.e., passive-standby, imposes periodic backups of the internal state for each service instance at a backup node. Passive-standby saves resources (e.g., bandwidth and storage) by storing the state of streaming instance at the backup node in regular intervals and not for every change of the internal state.

Periodic backups in passive-standby do not only help to save resources, they also facilitate seamless recovery of the data flow by redirecting the affected data streams to the backup node in the event of an original service instance failure. In case a failure occurs, the upstream service instance detects by means of periodic timeouts the disappearance of the downstream service instance and redirects the whole data stream to the backup node. Figure 4.2 illustrates the passive-standby recovery strategy.

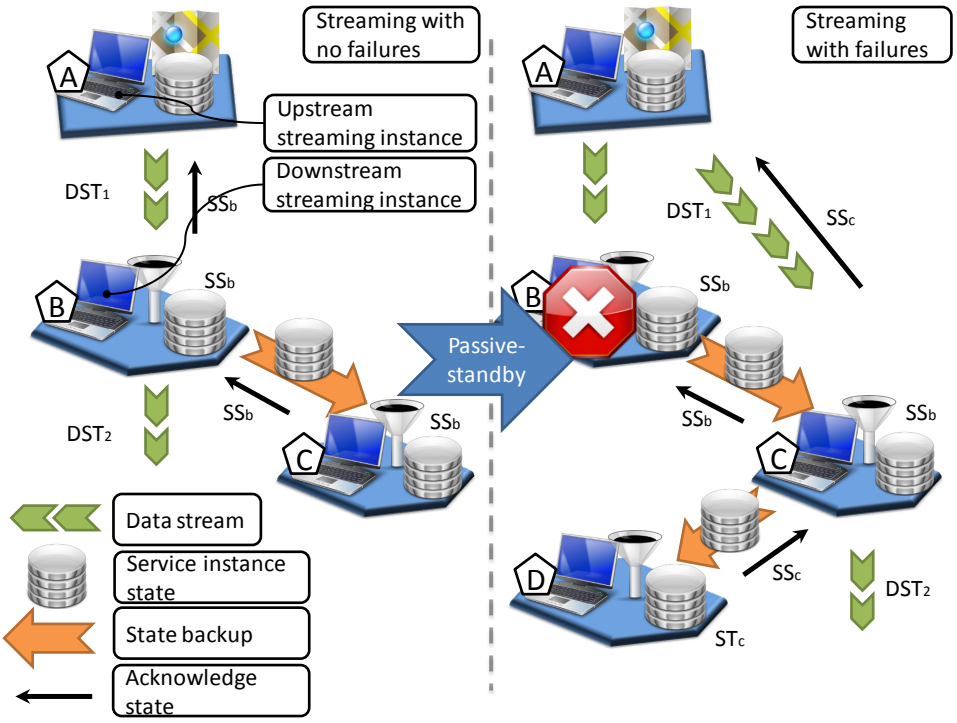


Figure 4.2: Continuous data flow with periodic backups that is subject to node failure.

Example 4.2

The example shown in Figure 4.2 depicts the passive-standby recovery of a data flow node failures. As the figure shows the nodes A, B and C are streaming instances. Thereby, node A is of different service type (depicted with the maps icon) than the nodes B and C (depicted with the sink icon). Being of continuous service type all nodes also feature an internal state (depicted with the cylinder icon). As illustrated node A is the upstream streaming instance of node B and sends a data stream (i.e., DST_1) of data items to it, whereas node C is the backup node for node B. In conjunction to the data stream DST_2 which node B sends downstream to some other node, it also sends periodic backups of its state SS_b (depicted with the orange thick arrow) to node C. In turn, node B acknowledges successful backups at (i.e., depicted with the black arrow) B to node A

In case of a failure of node B the data flow is recovered in the passive-standby sense by redirecting the data stream DST_1 to the backup node C. Based on the locally existing backed-up state SS_b at node C and the redirected data stream DST_1, node C resumes the streaming of the data flow DST_2. Moreover, node C finds its backup site of the same service type at the node D, which is thereafter provided with regular checkpoints of node C's state.

The longer the backup intervals are the more resources are saved, but also the longer it takes for the backup node to recover the state at which the original streaming instance failed. It is reasonable to assume that the backup will not feature the latest state of

the failed service instance at the time of a failure. Rather the difference of the backup node state and the state failed service instance will correspond to the number of data items that have been processed since the last checkpoint. The lost state can however be recovered by replying the state transitions at the backup node with the same data items as the failed service instance did. For this purpose the upstream service instance has to resend the missing data items to the backup node. In order to be able to know which data items to send, a downstream node has to acknowledge its checkpoint, i.e., the backed up state along with the last input data item, to the upstream node. Based on the checkpoint the upstream node can deduce the data items to be resent, and the backup can based on its state resume stream processing The exact backup interval duration is usually end-user application specific and is provided by means of ω_{ch}. Algorithm 4.1 shows the continuous data flow processing in conjunction to periodic check-pointing of the service instance state.

Algorithm 4.1 $si.dataFlow()$. Continuously reads all data items $(DI_{I1}, DI_{I2}, ..., DI_{In})$ for each input stream of I_c. Updates the internal state of the service instance si to ss_{new} and creates output data items $(DI_{O1}, DI_{O2}, ..., DI_{Om})$ for each output stream O_c. In case the checkpoint window ω_{ch} has been exceeded at current timestamp $\tau_{current}$ it performs the checkpoint at the backup service instance set SI_{ch}. For a successful backup it sends an acknowledgment message to each upstream service instance of SI_{up} that features a return value of the backup procedure and the last processed data item of each upstream service instance.

1: **while true do**
2: $(DI_{I1}, DI_{I2}, ..., DI_{In}) \leftarrow n.getDI(I_c)$
3: $((DI_{O1}, DI_{O2}, ..., DI_{Om}), ss_{new}) \leftarrow f_{st}((DI_{I1}, DI_{I2}, ..., DI_{In}), si.s.ss)$
4: $si.s.ss \leftarrow ss_{new}$
5: $n.putItem(O_c, (DI_{O1}, DI_{O2}, ..., DI_{Om}))$
6: **if** $\tau_{current} - \tau_{ch} \geq \omega_{ch}$ **then**
7: $retVal \leftarrow n.backup(ss_{new}, SI_{ch})$
8: $\tau_{ch} \leftarrow \tau_{current}$
9: **if** $retVal \neq$ **null then**
10: $SI_{up} \leftarrow n.getUpstreamNodes()$
11: **for** $i = 1$ **to** $|SI_{up}|$ **do**
12: $si_i \leftarrow SI_{up}[i]$
13: $si_i.acknowledge(DI_{Ii}, retVal)$
14: **end for**
15: **end if**
16: **end if**
17: **end while**

Note, that the backup routine `backup()` is in charge of replicating the latest service instance state to the backup service instance set SI_{ch}. In the traditional passive-standby sense this set features only one service instance (i.e., $|SI_{ch}| = 1$). Acknowledgment does not only help to deduce the missing data items for failure recovery, it also facilitates the detection of downstream node failures. In case the downstream service instance acknowledgments are delayed for a longer period of time, the service instance can as-

sume the downstream service instance to have failed. In such a situation the affected data stream can be redirected to the backup service instance and all data items starting from the last acknowledgment resent. Algorithm 4.2 shows the recovery of a downstream service instance induced by an missing acknowledgment timeout.

Algorithm 4.2 $si.ackTimeout(si_f)$ In the event of service instance si_f failure redirects its output stream DST_s to the backup service instance si_{bk}.

1: $DST_s \leftarrow si.getStream(si.s.ss, si_f)$
2: $\exists si_{bk} \in SI_{ch} : si_{bk}.s.ss = si_f.s.ss$
3: $si.setStream(si.s.ss, si_{bk}, DST_s)$

Observe from Algorithm 4.1 (i.e., line no.8), that the throughput of the service instance data flow depends on the efficiency of the backup process itself. In case the backup process is fast, the streaming instance will instantaneously address the input data streams so as to consume their data items and to transition its state. On the other hand, if the backup process is blocking due to the way it is implemented, then processing of the subsequent data items has to wait until the backup has been completed. Depending on the end-user application the service instances the routine `backup` can be implemented to be synchronous (i.e., blocking) or asynchronous (i.e., non-blocking) w.r.t. the data flow. Asynchronous backup processes feature high throughput of the data flow at the price of higher recovery time. That is, upstream service instances can only be acknowledged once the backup has been asynchronously completed thus causing a greater gap between the current and the backed-up state.

4.2 Workflow Execution Environment

The purpose of this section is to provide conceptual insights on the execution model for workflow definition on which our work is founded. First we elaborate the distribution concepts behind workflow engine system services. Afterwards, we shed light on the distributed data management concepts on which the system service distribution is built. Finally, we provide an illustrative example that demonstrates the execution of a workflow definition based on our approach.

4.2.1 Distributed Orchestration Service

As discussed in the motivation chapter (i.e., Chapter 2) of this thesis an ideal workflow engine implies scalability and resource conservation. Precisely, it should be possible to distribute the engine to the biggest available set of nodes (e.g., at all nodes in the execution environment) while consuming only very few memory, CPU and network resources at them.

So far the *Peer-To-Peer* approach has been identified as a convenient way to distribute critical functionalities (e.g., data management) among all available nodes so as to overcome the drawbacks of centralization. In the context of workflow instance execution, this implies that the overall orchestration workload of the engine has to be

collaboratively shared among all available nodes. That is, all nodes in the environment should be able to orchestrate workflow instances and thus invoke application service instance. Therefore, the global orchestration functionality is partitioned among all nodes in a Peer-to-Peer fashion by distributing the orchestration system service to them – this primarily includes the application service providers. The service type s_o is henceforth referred as the orchestration service and offers the functionality of distributed orchestration of workflow instances, such that it is deployed at nodes of interest inside an execution environment:

$$\exists s_o \in S \wedge \exists SI_o = \{si_o \in SI | \forall si_o \in SI \; si_o.s = s_o\}$$

Nodes hosting the orchestration service are referred to as the *orchestrator* nodes. The global orchestration functionality is partitioned in a way, which enables each node to orchestrate only locally available application services. Since in practice, no orchestrator is locally equipped with all possible application service types for all existing workflow definitions they have to collaborate by forwarding an instance of a workflow definition among each other. Thereby, only the orchestrator in possession of the instance can invoke the locally available application service instances and thus change its execution state whereas the other ones have to wait for their turn. Upon completion of the local application service instance invocations the control over the running workflow instance has to be transferred to another orchestrator node capable of advancing the execution.

This approach to orchestrator collaboration naturally fits the control flow execution semantics of our workflow definition model. The forwarding of a workflow instance among the distributed orchestration services at runtime follows (is bound to) the state transition semantics of a workflow instance. Whenever the active service instance set (SI_a) of a workflow instance changes, late-binding of orchestrators that locally host the currently active service instances will be performed. That is, they will receive the workflow instance from the previous orchestrator.

4.2.2 Orchestration Service Metadata

The distribution of the workflow engine among application service providers in a Peer-to-Peer fashion is feasible only if the corresponding system services are empowered with local data. In case of the orchestration service, for a node to be able to autonomously migrate workflow definition instances it has to be locally informed about the current state of the global execution environment. In particular, information on the currently available service instances, their host workloads, their host locations etc. is of the utmost interest to an orchestrator. Service instance host metadata empowers the orchestrators to make sound decisions as on where to forward the workflow definition instances at control flow migration time. In case this metadata is provided locally interference of a centralized authority is averted and thus scalability of the control flow migration is guaranteed. The formal definition of the service instance host metadata used for late-binding of the workflow instances is provided as follows:

Definition 4.10 (Service host metadata – h)**.** *The host metadata of a node $n \in N$ at timestamp $t \in T$ is defined as tuple $h(n,t) = l$ where l is current workload of node n with $0 \leq l \leq 100 \wedge l \in I\!R^+$.* □

Note, that H is disjoint based on the node identifier value n. A node can only feature one load and location address value at the same time. In order to prevent overload at the orchestrator nodes with excessive information on whole execution environment downsizing of the metadata volumes has to be performed. Thereby, nodes locally only maintain metadata that is relevant to their orchestration tasks. Selection of relevant metadata is performed on a need-to-know basis depending on locally available application service instances and the workflow structures that encompass them.

Precisely, given all workflow definitions structures, only execution environment metadata that characterizes remote service instance hosts which are direct successors to the locally available services instances are considered as relevant. In other words, only possible control flow successor host metadata is locally stored. Metadata on service instance hosts which will never be used for workflow instance migration, due to lack of subsequent application services, can be omitted. This way the amount of metadata at a peer can be significantly reduced. The actual amount of shipped data is dependent on the number of existing workflow definitions and the number of application service instances. For this to work, each peer additionally requires data on all workflow definitions. Based on this data a peer can autonomously determine the next step of the execution path for all succeeding application services.

An important aspect of the Peer-to-Peer metadata maintenance is its complete separation from the control flow. Peers should not have to query other peers, or some centralized authority for metadata at workflow definition instance migration time and thus slow down the orchestration process. Rather, the metadata should be locally available even before the migration happens so as to facilitate efficient (i.e., immediate) local migration decision making.

Separation of data management from the control flow facilitates scalability of the distributed orchestration service as arbitrary numbers of workflow definition instances can easily be ceded to the orchestrator nodes for autonomously steering of their executions. In order to empower the nodes with the most necessary metadata in a timely fashion data replication has to be applied. The data replication should operate in concurrence to the control flow migration, and continuously update the node metadata for any change of it. Given the continuous metadata replication process, the distributed orchestration service at a peer can rely upon the locally available metadata whilst deciding on where to migrate the workflow definition instances.

To keep the data replication process free of uncontrolled flooding, and thus efficient, our approach is based on a publish-subscribe (i.e., pub/sub) replication scheme. Orchestrator nodes issue subscriptions with a clear intent for interested host metadata, that is based on workflow definition precedence orders. Precisely, for a given workflow definition orchestrator nodes only subscribe for metadata of hosts that feature successor activities for the locally hosted service types. In turn, they are supplied with the subscribed metadata whenever there is a significant update to it. For the pub/sub scheme to work all orchestrator nodes also have to publish significant changes of their metadata, which is further propagated for any existing subscription to them.

Subscription creation is only a feature of the orchestration service functionality. Each orchestration service enabled node, has to create subscriptions for all of its locally hosted service instances, such that all workflow definitions, that could request orchestration

decisions from it, are covered. The formal definition of a subscription is provided as follows:

Definition 4.11 (Metadata subscriptions – sub)**.** *A subscription for execution environment metadata, issued by the orchestrator service instance $si_o \in SI_o$, is defined as tuple $sub(si_o) = (wf, si_l, A_{succ})$ where:*

- *$wf \in WF$ is a workflow definition,*
- *si_l is the local service instance for which the subscription is issued by si_o:*

$$si_l \in SI \wedge si_l.n = si_o.n$$

- *A_{succ} is the activity subset of $wf.A$ that si_o subscribes to such that service instances featuring activities of A_{succ} succeed si_l, w.r.t. control flow, with:*

$$A_{succ} = \{a \in wi.A | \forall a \in wi.A \; \exists (si_l.s, a) \in wf. \prec_{cf}\}$$

The set of all subscriptions issued by one orchestrator service instance is denoted as $SUB(si_o)$, whereas the set of all subscriptions for all orchestrator service instances SUB. □

The global metadata repository maintains all information that is necessary for autonomous control flow migration. The specific sets that are necessary so as to power the orchestration service consists of the workflow definition set WF, the service instance set SI, the subscription set SUB and the hosts set H. That is, the global metadata repository aggregates all existing workflow definitions, all existing service instances, all metadata subscriptions for all nodes and all current workloads for all nodes. The formal definition of the repositories is provided as follows:

Definition 4.12 (Global metadata repository – ρ)**.** *The global repository ρ is defined as union of the workflow definition set WF, the service instance set SI, the subscriptions set SUB and the hosts metadata set H with:*

$$\rho = WF \cup SI \cup SUB \cup H$$

. □

By default all service orchestration nodes of an execution environment are always subscribed to the global metadata repository ρ. The mandatory subscription to ρ enables the orchestrator nodes to learn about new workflow definitions and to consequently deduce new successor application activities they are obliged to subscribe for.

$$\forall si_o \in SI \textbf{ where } si.s = s_o \implies SUB(si.n) \neq \emptyset$$

Whenever a new workflow definition is added to the repository subscriptions are created based on control flow precedence orders. Precisely, each service instance is provided with subscriptions on other service instances that are succeeding it in the precedence order of the control flow of the newly added workflow definition. Only metadata on the subscribed succeeding service instances thus needs be replicated to the orchestrator nodes. Algorithm 4.3 describes in detail the subscription creation in the event of a new workflow definition addition.

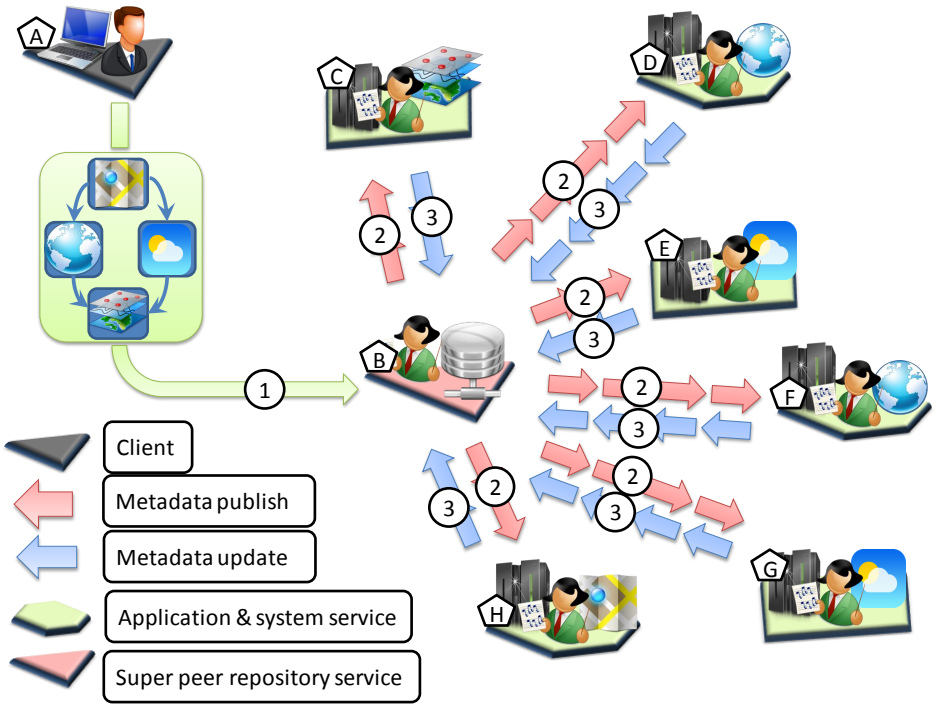


Figure 4.3: Repository metadata exchange.

Control flow based subscriptions imply that orchestrator nodes rely on parts (i.e., subsets) of the global repository metadata. That is, orchestrator are only interested in a subset of the global metadata repository ρ that reflects the current state of the succeeding service instances for a given subscription. This metadata subset is locally stored at the orchestrator nodes so that they can manage workflow instance runtime without having to query the repository. Formally, the orchestrator node local metadata is defined as follows:

Definition 4.13 (Orchestrator service metadata – md)**.** *Execution environment metadata md locally maintained by the orchestrator si_o is defined by the tuple $md(si_o) = (sub_l(si_o), \rho_h)$ such that:*

- *$sub_l(si_o) \in SUB$ is the subscription issued by the orchestrator si_o on behalf of a local service instance,*
- *ρ_h is the locally maintained subset of the global repository, w.r.t. the local subscription $sub_l(si_o)$, with:*

$$\rho_h = \{h(n,t) \in H | \forall a_{succ} \in sub_l(si_o).A_{succ} \exists (a_{succ}, n) \in SI\}$$

The set of all execution environment metadata for one orchestrator service instance is defined as $MD(si_o)$, whereas the set of all metadata for all orchestrator service instances is defined as MD. □

The orchestrator service metadata is subject to continuous updates by means of the pub/sub replication mechanism. Whenever a subscription is issued to the subscriptions

Algorithm 4.3 $\rho.addWf(wf)$ creates subscriptions for all service instances of SI on succeeding service instances for the control flow of wf, in the global repository ρ.

1: $WF \cup wf$
2: **for** $si \in SI$ **do**
3: $A_{succ} = \{a_{succ} \in wf.A | \forall a_{succ} \in wf.A(si.s, a_{succ}) \in wf. \prec_{cf}\}$
4: **if** $A_{succ} \neq \emptyset$ **then**
5: $sub((s_o, si.n)) \leftarrow (wf, si, A_{succ})$
6: $SUB \cup sub$
7: **for all** $a \in A_{succ} : a.isSecondary$ **do**
8: **if** $a.isSecondary()$ **then**
9: $\rho.subscription(sub((s_o, si.n)), \textbf{false})$
10: **else**
11: $\rho.subscription(sub((s_o, si.n)), \textbf{true})$
12: **end if**
13: **end for**
14: **end if**
15: **end for**

repository, the hosts metadata has to be selected that is featured by the subscribed service instances. Subsequently, the selected hosts metadata is published to the subscribed orchestrator node. Algorithm 4.4 shows the metadata publishing routine.

Algorithm 4.4 $\rho.subscription(sub(si_o), forward)$ selects metadata out of the hosts set H that is featured by the service instances of the given subscription sub. Publishes the metadata to the orchestrator service instance of sub if specified in $forward$.

1: **if** $forward$ **then**
2: $Hosts = \{h(n,t) \in H | \forall h(n,t) \in Hosts \; \exists a_{succ} \in sub(si_o).A_{succ} : (a_{succ}, n) \in SI\}$
3: $md(si_o) \leftarrow (sub(si_o), Hosts)$
4: $si_o.replicate(md(si_o))$
5: **end if**

The routine shown in Algorithm 4.4 is not only executed whenever workflow definitions are added. In case there is a significant change in the services instance set (i.e., newly added or removed service instances) or the hosts set (i.e., updated workload information) that affects the already existing subscriptions, the routine `subscription()` is executed.

The pub/sub scheme replication mechanism asserts another important system service, that is the pub/sub repository service s_ρ, along with the orchestration service s_o. So far s_ρ and s_o constitute the distributed system services that can be deployed to any peer of the execution environment. Following the publish/subscribe replication schema the repository itself can subscribe to other repositories (if existent) in a recursive manner. This way hierarchical repository trees can be spanned that in conjunction with intelligent subscription configuration can provide powerful means of Peer-to-Peer data propagation. Further details on the publish/subscribe replication mechanism can be found in [SST+05, STS+06].

Example 4.3

Figure 4.3 illustrates an example of the metadata propagation among the peers of an execution environment. As the figure shows, the execution environment is composed of eight nodes (i.e., A, B, C, D, E, F, G and H) out of which only one is a simple client node (i.e., A), whereas the other ones are system nodes. Out of this set of system nodes, only node B is responsible for hosting the metadata repository ρ and is thus the repository peer (i.e., displayed with the red service color and the database icon). The other system nodes are in charge of hosting the orchestration service along with their corresponding application services (i.e., displayed with the green service color, the orchestrator icon and the corresponding application service icon).

The workflow structure consists of four application services types, i.e., a heat-map, a map, a weather forecast and a location service type. The structure of the workflow definition corresponds to the one of Figure 4.1. However, the smaller, dashed and red and blue arrows are new to this example and represent the metadata exchange among the system nodes with the repository peer. The red arrows correspond to metadata being published to the orchestrator service at the system nodes. Whereas the blue arrows correspond to the metadata changes being updated at the repository by the system nodes. The interaction sequence is represented with the numbered white circles on top of the interaction arrows. Regarding the service instances SI_{app}, the execution environment is composed as follows:

$$SI_{app} = \{(s_2, H), (s_3, E), (s_3, G), (s_4, D), (s_4, F), (s_5, C)\}$$

Regarding the orchestration service instances, the environment is composed as follows:

$$si_b = (s_o, B) \ \wedge \ si_c = (s_o, C) \ \wedge \ si_d = (s_o, D) \ \wedge \ si_e = (s_o, E) \wedge$$

$$si_f = (s_o, F) \ \wedge \ si_g = (s_o, G) \ \wedge \ si_h = (s_o, H)$$

$$SI_s = \{(s_{rep}, B), si_b, si_c, si_d, si_e, si_f, si_g, si_h\} \textbf{ where } SI = SI_{app} \cup SI_s$$

Whenever a workflow definition $wf \in WF$ enters the system (by a client execution request) it is in a first step forwarded to the repository ρ hosted at the super peer, i.e., $\rho.addWf(wf)$. Since all system nodes are at all times subscribed to the repository ρ they have to be updated (i.e., step no.2) with the latest metadata in the next step with succeeding service instances. That is, for each system node subscriptions have to be created that feature succeeding service instances. Precisely, node H subscribes for the map and weather forecast services as they are succeeding the location service according to wf with:

$$sub(si_h) = (wf, \{(s_2, H)\}, \{(s_3, E), (s_3, G), (s_4, D), (s_4, F)\})$$

Jointly, nodes D, E, F and G subscribe for the heat-map service with:

$$sub(si_d) = (wf, \{(s_4, D)\}, \{(s_5, C)\}) \ \wedge \ sub(si_e) = (wf, \{(s_3, E)\}, \{(s_5, C)\}) \wedge$$

$$sub(si_f) = (wf, \{(s_4, F)\}, \{(s_5, C)\}) \wedge sub(si_g) = (wf, \{(s_3, G)\}, \{(s_5, C)\})$$

The newly created subscriptions are added to the subscription set at ρ by means of `subscription()` routine. This causes the selection and replication of hosts metadata to system nodes. Node H receives hosts metadata on nodes G, E, D and F, whereas E and G receive metadata on C with:

$$md(si_h) = (sub(si_h), \{10.01, 20.01, 40.01, 30.01\} \wedge$$

$$md(si_e) = (sub(si_e), \{50.01\} \wedge md(si_g) = (sub(si_g), \{50.01\})$$

Nodes D and F will not receive metadata as they are their application services are secondary activities. Secondary activities will be described in the next section. In the final step (i.e., step no.3) system nodes start to update the repository with metadata changes so that the other subscribed system nodes can be informed. The propagation of node metadata to the repository and back goes on in the background even after the subscription creation has finished and continuously updates the local metadata at nodes for any state change of the subscribed nodes.

Finally, the applied approach to data management does not come without a price. As the peer metadata accuracy is solely dependent on the propagation performance of the repository service node it might feature (e.g., overload situations) staleness issues at times. However, since the maintained data merely reflects the state of the execution environment this should not lead to incorrect control flow migration in general. Rather, it could lead to suboptimal migration decisions based on the current metadata on the execution environment (e.g., stale workload node metadata), which is in the case of the distributed orchestration service acceptable. More severe inaccurate metadata situations (e.g., stale service instance metadata) should be occurring less frequently and can be handled at control flow migration time with retries.

Distributed Workflow Definition Execution

Only when all orchestrator nodes have been locally provided with metadata the distributed execution of an instance of a workflow definition can commence. Based on the locally available metadata, orchestrator nodes can autonomously make decisions as on how to steer the execution of a workflow instance, i.e., how to perform late-binding. That is, whenever the invocation of local service instances has been completed the orchestrator decides to which succeeding service instances should be late-bound and the workflow instance be forwarded to.

In order to late-bind an activity to a service instance, the orchestrator has to chose a subsequent activity out of the set of subsequent activities A_s such that it succeeds the finished invocation activity w.r.t. the control flow. Based on the locally available hosts metadata, the orchestrator chooses a service instance whose host features the best runtime characteristics for some end-user application requirement. If not specified explicitly, the orchestrator will always late-bind the service instance that features the least current workload with the workflow instance. Algorithm 4.5 shows the optimal peer selection with regard to minimal workload.

Algorithm 4.5 $n.getSI(a_{next})$ Finds the metadata that reflects the given activity a_{next}. Selects the node with the minimal workload out of the hosts metadata subset ρ_h and returns it as a optimal service instance.

1: $\exists md(s_o, n) \in MD(s_o, n) : a_{next} \in md.A_{succ}$
2: $\exists (l_{opt}, t_{opt}) \in md(s_o, n).\rho_h : \forall (l_x, t_x) \in md(s_o, n).\rho_h \; n_{opt} \neq n_x \wedge l_{opt} < l_x$
3: **return** (a_{next}, n_{opt})

Once the optimal succeeding service instance has been picked the orchestrator can migrate the workflow instance to it so as to continue the execution. To complete the migration the orchestrator has to update the workflow instance just before sending it. The orchestrator has to change the state of the workflow instance by shifting the subsequent activity from the subsequent activity set A_s to the currently active set A_a and by adding the optimal succeeding service instance to the activated service instances set SI_a. In case the completed invocation was of discrete service type the completed activity has to be consequently shifted from currently active set A_a to the completed activity set A_c and the finished service instance has to be removed from the activated service instances set SI_a and added to the competed activity set SI_c. Activities of streaming type usually do not finish their execution with one invocation, thus they do not need to be removed from the active set A_a of a workflow instance. Algorithm 4.6 shows the standard workflow instance migration routine.

If parallel activities exists in the control flow their invocation results might be subject to merging at some subsequent join activity. In order for parallel orchestrator nodes to migrate their workflow instances in a Peer-to-Peer fashion, they have to agree on one service instance to accept all parallel workflow instances. Parallel orchestrator node agreement is achieved by entrusting one parallel activity with the responsibility of selecting the joining service instance. This responsible activity is referred to as the *primary* activity. The other parallel activities are referred to as the *secondary* activities. The orchestrator node invoking a primary activity migrates its workflow instance in the standard migration routine sense, by selecting the optimal successor and forwarding it the workflow instance. To inform the secondary service instances about the optimal joining service instance the primary exploits the repository at runtime so as to propagate its selection to them. That is, the primary creates at runtime execution metadata, i.e., the information on the joining service instance, and stores this metadata at the repository. The joining metadata is formally defined as follows:

Definition 4.14 (Join metadata - j)**.** *Joining service instance metadata is defined by the tuple* $j = (wi, a_j, si_j)$ *where:*

- $wi \in WI$ *is the workflow instance to be joined,*
- $a_j \in wi.wf.A$ *is the join activity,*
- $si_j \in SI$ *is the service instance at which wi is joined with:*

$$si_j.s = a_j$$

The set of all joining service instance metadata is denoted as J. □

Consequently, the joining metadata set J is also maintained by the global metadata repository with:

$$\rho \cup J$$

Hence, just before the orchestrator of a primary activity migrates its workflow instance it updates the repository ρ with the joining service instance metadata. On the other hand, the secondary activity orchestrator nodes wait for an update from the repository. Once the secondaries receive the joining service instance metadata they can migrate their workflow instances towards it and not before that.

Algorithm 4.6 $si_o.migrate(wi)$ Invokes all local service instances. Updates the given workflow instance wi state by removing the invoked service instances of discrete type and their activities from it. Afterwards it finds the optimal succeeding service instances for the subsequent activities and migrates updated wi, in terms of A_a, A_s, SI_a and SI_c, to them. In case the subsequent activity is parallel, it updates the repository ρ with the optimal succeeding service metadata.

1: **for** $si \in wi.SI_a : si.n =$ **this do**
2: $si.invoke(wi)$
3: **if** $si.a \in S_d$ **then**
4: $wi.A_a \setminus si.a$
5: $wi.SI_a \setminus si$
6: $wi.A_c \cup si.a$
7: $wi.SI_c \cup si$
8: **end if**
9: $A_{next} = \{a_{next} \in wi.A_s | \forall a_{next} \in wi.A_s \ (si.s, a_{next}) \in wi.wf. \prec_{cf}\}$
10: **for** $a_{next} \in A_{next}$ **do**
11: **if** $\neg a_{next}.isSecondary()$ **then**
12: $si_{next} \leftarrow n.getSI(a_{next})$
13: $wi.A_s \setminus a_{next}$
14: $wi.A_a \cup a_{next}$
15: $wi.SI_a \cup si_{next}$
16: $si_o \leftarrow (s_o, si_{next}.n)$
17: **if** $a_{next}.isPrimary()$ **then**
18: $si_o.migrateJoin(wi, a_{next})$
19: $\rho.addJoin(wi, a_{next}, si_{next})$
20: **else**
21: $si_o.migrate(wi)$
22: **end if**
23: **end if**
24: **end for**
25: **end for**

Although being subscribed at ρ for succeeding activities, that are joining, secondary activity orchestrators will not be updated with metadata initially. There exists no need to update the secondary activities, with potential successors at subscription creation time, as they will never get to choose a succeeding service instance. Hence, whenever a

new workflow definition is added and consequently subscriptions are created, the secondary activities are omitted from metadata publishing (i.e., Algorithm 4.3, line no.9)). When the primary activity orchestrator informs the repository about the joining service instance the secondary orchestrator nodes will be updated with this information, as well. However, the secondary orchestrator nodes will be updated with metadata of only one succeeding host. Afterwards, the secondary orchestrator nodes will be instructed to do the join of their workflow instances by means of the routine `doJoinMigration()`. Algorithm 4.7 shows the joining service instance addition to the repository.

Algorithm 4.7 $\rho.addJoin(wi, a_j, si_j)$ Adds the join service instance to the join metadata set J. Selects all subscriptions that are subscribed for the given join activity a_j. Updates the subscribed orchestrator nodes with hosts metadata on given the join service instance si_j.

1: $J \cup (wi, a_j, si_j)$
2: $Subs = \{sub(si_o) \in SUB | \forall sub(si_o) \in SUB \ \exists a_j \in sub(si_o).A_{next} \wedge sub(si_o).si_l.isSecondary() = \textbf{true}\}$
3: **for** $sub(si_o) \in Subs$ **do**
4: $\quad \exists h(n,t) \in H : si_j.n = n$
5: $\quad si_o.replicate(si_o, sub(si_o), h(n,t))$
6: $\quad si_o.doJoinMigration()$
7: **end for**

Once the secondary orchestrator nodes have migrated their workflow instances to joining service instance the they can be merged into one. The end-user specific merging of parallel workflow instances is performed by means of the routine `merge()`. Consequently, the merged workflow instance can be sent to the local joining activity for invocation. Algorithm 4.8 shows the joint workflow instance migration.

Algorithm 4.8 $si_o.mirgateJoin(wi, a_j)$ Aggregates all parallel workflow instances wi. When all have been received merges them into one and forwards it for invocation.

1: $JoinWis[a_j] \cup wi$
2: $Preds = \{a \in wi.wf.A | \forall a \in wi.wf.A \ \exists (a, a_j) \in wi.wf. \prec_{cf}\}$
3: **if** $|JoinWis[a_j]| = |Preds|$ **then**
4: $\quad wi \leftarrow n.merge(JoinWis[a_j])$
5: $\quad si_o.migrate(wi)$
6: **end if**

Now that we have elaborated in detail the concepts behind the distributed workflow instance execution we bring them together with an illustrative example in Figure 4.4. The displayed example plays through the execution of the already introduced workflow definition $wf \in WF$ of Figure 4.3 in a step-by-step fashion.

Example 4.4

As we can observe, the execution environment in Figure 4.4 is composed in the exact

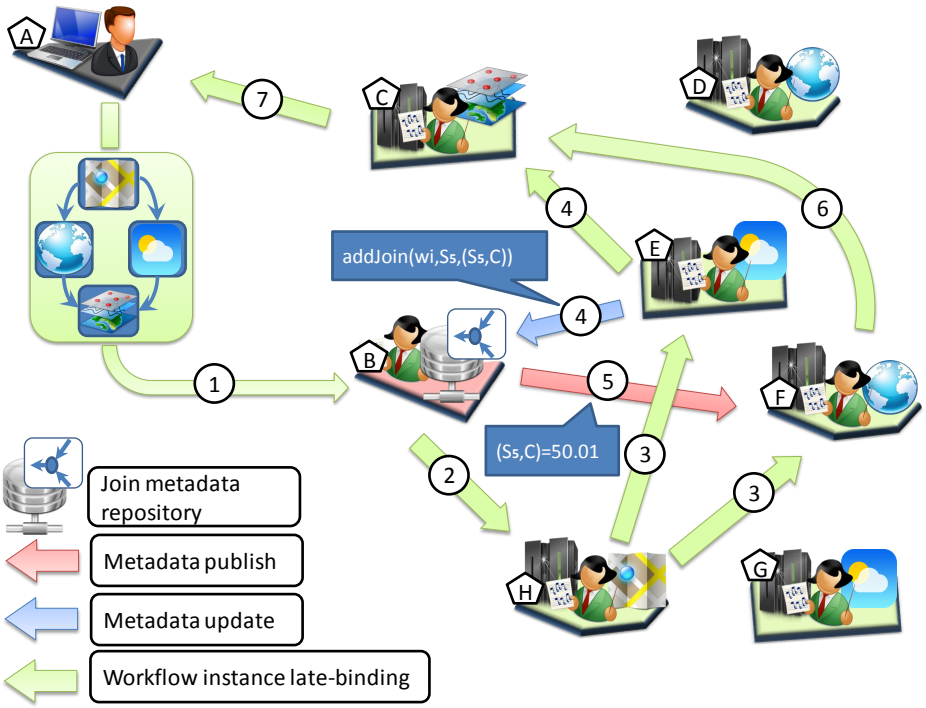


Figure 4.4: Distributed orchestration service execution model.

same way as in Figure 4.3. This example assumes however, does not cover the initial metadata exchange as in Figure 4.3 so that actual execution can start. Hence, the workflow definition execution is conducted as follows:

1. Whenever a workflow definition $wf \in WF$ enters the system (by a client execution request) it is in a first step forwarded to the workflow definition set at ρ hosted at the super peer and the metadata is exchanged.

2. The super peer creates then an instance of the corresponding workflow definition $wi \in WI$ and forwards it to the first node capable of advancing the execution, by means of the `migrate(wi)` routine. In this example only node H applies for the fist execution step. The internal state of the instance wi is updated accordingly:

$$wi = (wf, \underbrace{\{\emptyset\}}_{A_c}, \underbrace{\{s_2\}}_{A_a}, \underbrace{\{s_3, s_4, s_5, a_e\}}_{A_s}, \underbrace{\{\emptyset\}}_{SI_c}, \underbrace{\{(s_2, H)\}}_{SI_a})$$

   The currents state of the instance wi of timestamp 1 is also reflected in the workflow instance history, i.e., $wi_1 = (1, wi) \in WI_t$. When node H receives wi it assumes control over the execution of the workflow instance and contributes to it by invoking the locally available hat map service.

3. Once node H has finished with the invocation of its application service instance it advances the execution of wi in this step by migrating it to the succeeding application service instance. Since the next step mandates two succeeding application service types (i.e., the map service and the weather forecast service) node H sends

in parallel two instances to the best corresponding providers, i.e., nodes E and F. The other providers of those service feature a higher individual workload so they are overlooked for migration. Hence, workflow instance is updated at timestamp 2 to $(2, wi) \in WI_t$ where:

$$wi = (wf, \underbrace{\{a_s, s_2\}}_{A_c}, \underbrace{\{s_3, s_4\}}_{A_a}, \underbrace{\{s_5, a_e\}}_{A_s}, \underbrace{\{(s_2, H)\}}_{SI_c}, \underbrace{\{(s_3, E), (s_4, F)\}}_{SI_a})$$

4. In the fourth step of the execution the parallel invocations at nodes E and F have to be merged into one. In our case, the weather retrieval application service branch (i.e., node E) is selected to be the primary activity branch. Once node E has finished the invocation of its application service type it selects the succeeding application service instance (i.e., node C) and migrates its instance copy to it at timestamp 3 to $(3, wi) \in\in WI_t$ where:

$$wi = (wf, \underbrace{\{a_s, s_2, s_3\}}_{A_c}, \underbrace{\{s_5, s_4\}}_{A_a}, \underbrace{\{a_e\}}_{A_s}, \underbrace{\{(s_2, H), (s_3, E)\}}_{SI_c}, \underbrace{\{(s_5, C), (s_4, F)\}}_{SI_a})$$

Moreover, it additionally informs the repository ρ about the fact that this instance of the workflow definition is joined on node C, i.e., $J \cup (wi, s_5, ((s_5, C)))$.

5. The secondary activity node F eventually gets updated by the repository ρ with this information:

$$F \leftarrow md(si_f) = (sub(si_f), (50.01))$$

6. Consequently, node F migrates wi to node C at timestamp 4 to $(4, wi) \in WI_t$ where:

$$wi = (wf, \underbrace{\{a_s, s_2, s_3, s_4\}}_{A_c}, \underbrace{\{s_5\}}_{A_a}, \underbrace{\{a_e\}}_{A_s}, \underbrace{\{(s_2, H), (s_3, E), (s_4, F)\}}_{SI_c}, \underbrace{\{(s_5, C)\}}_{SI_a})$$

In turn, node C migrates the received parallel workflow definition instances and invokes its application service.

7. In general, steps $no.3$ through $no.6$ are iteratively repeated for as long as there are activities to be invoked within the scope of the workflow definition. However, since step seven is the final step for this example and there are no more activities to be invoked, the execution of this instance completes with the forwarding of the results overall execution to the client that issued the request at timestamp 5 to $(5, wi) \in WI_t$ where:

$$wi = (wf, \underbrace{\{a_s, s_2, s_3, s_4, s_5\}}_{A_c}, \underbrace{\{a_e\}}_{A_a}, \underbrace{\{\emptyset\}}_{A_s}, \underbrace{\{(s_2, H), (s_3, E), (s_4, F), (s_5, C)\}}_{SI_c}, \underbrace{\{\emptyset\}}_{SI_a})$$

Depending on the executional duration of the parallel activities, their service instance invocations tend to finish at different times. In case secondary activity branches

complete their invocations before the primary activity they are confined to waiting till they get updated with the latest joining service instance information. This is why in practice, the activity with the smallest execution time has to be manually (by estimation) chosen to be the primary activity. Finally, in this example we have omitted the flow of data during the course of execution for simplicity reasons. However, as mentioned in the previous sections the data flow of a workflow definition is stored within the instance *wi* for discrete service types.

4.3 Summary

In this chapter we have introduced a formal model of distributed execution is based on workflow definitions that feature arbitrary combinations of service types. The comprised service types can be of any type, i.e., discrete or continuous. The only prerequisite is that they are structured in a well-formed precedence order of their invocations. This is also referred to as the control flow. A well-formed workflow defines the order in which the data is exchanged among the service types such that is coincides with the control flow. This is also referred to as the data flow.

The introduced formal model of distributed execution mandates that at runtime service instances of specified service types by workflow definition are invoked according to the control flow and supplied with data according to the data flow. This is also referred to as late-biding of service instances at runtime. The currently invoked service instances represent the execution state of the workflow definition. This state is subject to constant change in time and is conveyed by means of an instance of the workflow definition.

To perform late-binding of service instances orchestrator nodes are introduced within the context of the formal model. That is the orchestration system service is introduced that enables the hosting nodes to perform late-bindings w.r.t. precedence orders of workflow definitions. The orchestrator nodes autonomously migrate the workflow instance among each other according to the control flow, thus changing its state. In order to be able to do so, orchestrators rely on metadata on the execution environment. In particular metadata on subsequent service instances according of the control flow. This metadata is supplied to them from the repository system service in a pub/sub fashion which is decoupled from the control flow.

The introduced formal model offers two main benefits: load balancing and scalability. An orchestrator can choose the best subsequent service instance at runtime and the orchestration service can be deployed to an arbitrary number of nodes.

5

Self-organizing Workflow Execution Engines

In the previous chapters of thesis we have thoroughly discussed distribution concepts for data and workflow definition management. These concepts have been observed from a point of view of an ideal execution environment setting. However, workflow definitions require systems that manage the executions of their instances no matter what the circumstances in the execution environments are. The purpose of this chapter is to shed light on the distributed workflow management model in certain situations which are far from ideal. This includes service instance failures, highly heterogeneous execution environments, poor application service allocations at devices etc. In this chapter we merge the approaches to data and workflow management distribution in a conceptually novel fashion, by means of novel system services, that is driven by the goal of providing an ideal workflow instance execution.

First, we will set the stage by elaborating the distributed execution model for workflow instances, in face of pervasive node failures inside an execution environment. Then we will provide a fault-tolerance concept for the control flow that addresses node failure situations in the form of a self-organizing system service that is referred to as the Safety-Ring. The Safety-Ring service represents the most important contribution of this thesis. Moreover, the workflow execution model will be applied to domains that feature a high degree of heterogeneous nodes. To address heterogeneity in the context of a reliable control flow, novel concepts, i.e., Compass, are introduced that adapt the Safety-Ring service even to such environments. Since the workflows are also characterized by a flow a data we will expose the traditional reliability mechanisms to redundancy concepts in the form of the redundant passive-standby strategy. Finally, we introduce concepts that additionally improve the workflow execution model, in terms of improved instance throughput, in dynamically allocating service types at the most suitable nodes in the environment by means of a concept that features self-optimizing deployments of services.

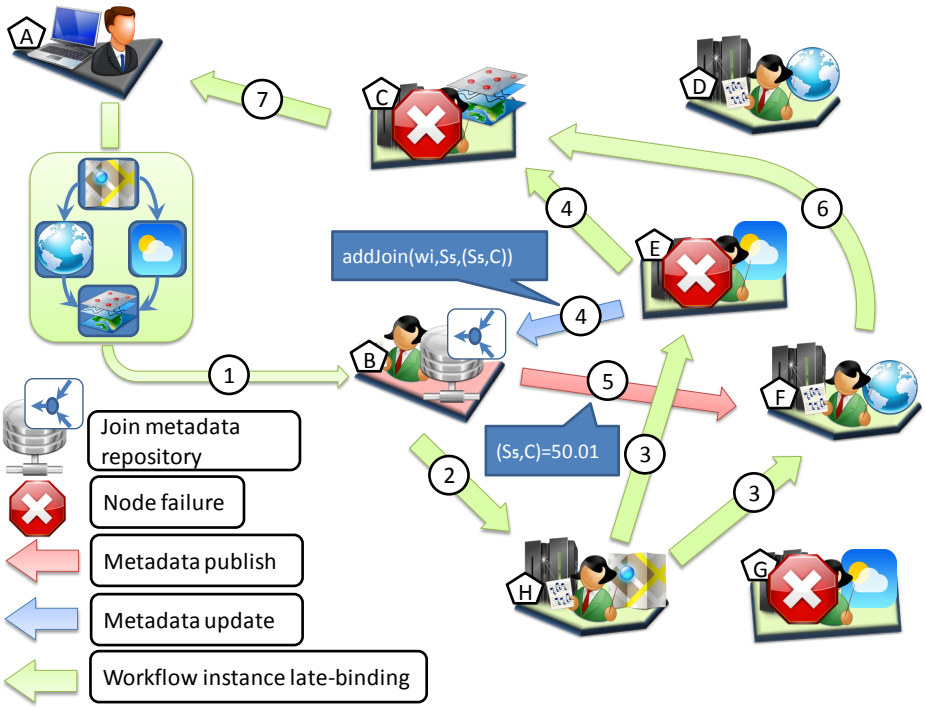


Figure 5.1: Distributed execution model with node failures

5.1 Self-healing Execution of Workflow Definitions

The workflow definition model introduced in Section 4.2 assumes execution conditions that are ideal, i.e., an environment in which nothing goes wrong. This assumption cannot always hold in dynamic environments that are concealed behind SOA and that feature a high degree of scalability (i.e., Clouds) or heterogeneity (e.g., Peer-to-Peer). In practice, it is reasonable to expect that in highly distributed environments participating nodes suddenly disappear due to internal hardware failures, resource depletion, or external damages etc.

Example 5.1

Figure 5.1 illustrates such an example in which the workflow definition model (as already introduced in Figure 4.4) is subjected to node failure issues. This implies that our example builds upon the same configuration of executing service instances and their exchanged metadata. Moreover, the execution steps bear the same meaning as in the example of Figure 4.4).

Depending on the current state of a running workflow instance *wi* w.r.t. to the control flow the failure of service instances might have severe effects on the further execution. Thereby, we distinguish between the featured activity types by the failed service instances w.r.t. the state of *wi* as follows:

- Completed activity sets failures – A_c, SI_c. If for example service instances fail that feature activities of the completed activity set in the workflow instance (i.e., $wi.A_c$) then such failures should not have any effects on the further control flow of wi. Unless advanced execution models are applied (e.g., based on semantic recovery) completed service instances of SI_c should not be traversed again in the course of a workflow instance migration. Therefore, their failure is insignificant from a late-binding point of view. In the context of semantic recovery model however, the failure of finished service instance makes the resolution of alternative compensating paths not possible, thus invalidates semantic recovery.

- Succeeding activity set failures – A_s. In case service instances fail that feature activities of the succeeding activity set (i.e., $wi.A_s$) in the workflow instance (e.g., node G before step no.3) then such failures should not have effects on the control flow whatsoever. The applied late-binding of activities will always resort to the remaining service instances of the same type. In case there are no service instances of the subsequent activity type (e.g., failure of node C), then the execution of the workflow instances cannot proceed and has to abort. Hence, late-binding of activities secures workflow instance migration from subsequent service instance failures.

- Active sets failures – A_a, SI_a. If service instances fail that feature activities of the active activity set (e.g., node C at step no.6) in the workflow instance (i.e., $wi.A_a$) and thus are invoked for execution (i.e., $wi.SI_a$) at the moment of failure than the effects are critical. With the failure of a active service instance the execution is permanently lost and so are the invocation results up to this point. As such service instances are in possession of the workflow instance wi their failure will cause wi to vanish. Being the control flow token that enables an orchestrator to invoke application service instances, the loss of wi will break the migration of the control flow and thus the execution will never conclude.

The aforementioned failure scenarios assume node failures of permanent type. That is, once failed the nodes do not come back ever again to contribute to the execution in the same context as of before the failure. On the other hand, temporary failures of service instances are not as troublesome as permanent ones. In most cases temporary failures can be easily overcome by means of workflow instance backups to local persistent storage. The backed-up workflow instances can be replayed (re-invocation of local application service instances) at the restart of the temporary failed orchestrator, in case the failure affects active service instances. Therefore, with respect to all possible failure scenarios, we conclude the distributed execution model based on workflow instance migration to be critically vulnerable to node failures of permanent type.

5.1.1 The Safety-Ring

In order to prevent losses of workflow instances and thus the interruption of the control flow they have to be made redundantly available. The most straightforward approach to workflow instance availability is permanent storage across a set of back up nodes

by means of timely replication. Whenever the execution state of a workflow instance changes it should be redundantly stored. Moreover, the service instance currently in control of the workflow instance, i.e., currently carrying out the invocation of active service types, has to be monitored so as to detect its failures. For each active service instance, there has to exist a recovery node that monitors it, and in the event of failure recovers it.

The set of monitoring nodes that recover failures of service instances constitutes a failure-recovery mechanism. Since the workflow engine is bounded to a set of requirements (e.g., Reliability – Section 2.4) so as to be considered ideal, we have to introduce a failure-recovery mechanism into the workflow engine. Thereby, the failure-recovery mechanism has to be designed to meet the requirements of an ideal workflow engine as well. Otherwise the distributed workflow execution model would be implicitly affected. To this end, we subject the failure-recovery mechanism itself to the following requirements, that are specific to the reliability requirement of an ideal workflow engine. The failure-recovery mechanism of an ideal workflow engine should feature efficiency, scalability, reliability and consistency as follows:

- Efficiency – The overhead of the recovery mechanism, that is reflected in the redundancy of backup data, should be minimal so as affect the execution of running workflow instances as least as possible. Moreover, the failure of service instances should be detected in a timely fashion. Whereas the handling of failures should feature fast recovery times such that the execution of affected workflow instances can be resumed as fast as possible. Moreover, its applicability on resource limited devices should also be an option.

- Scalability – The increase in available service instances and concurrently running workflow instances in the execution environment should not affect efficiency of recovery. Rather increase of service/workflow instances should be accommodated with the increase of recovery nodes in a scalable fashion by the recovery mechanism. Thereby, the overhead of the recovery mechanism (i.e., data redundancy and assignment of monitoring responsibilities) should be evenly distributed among the recovery nodes. With the change of workflow instances and service instances numbers the load and the number of recovery nodes should be automatically balanced.

- Reliability – The recovery nodes should be reliable themselves. In a Peer-to-Peer execution environment no node can be assumed to be stable at all times, not even the recovery nodes. The failure of a recovery node, should be recovered as efficiently as the failure of a service instance itself. Hence, each recovery node should be organized redundantly across a set of nodes regarding its failure-recovery responsibilities. Given that recovery of service instances is limited by the availability of the substitution service instances of the same type, this should not be the case for recovery nodes. It should be possible to organize the recovery nodes redundantly at any distributed workflow engine node.

- Consistency – The workflow instances of the same definition and the same input data should finish with the same result despite of any failure at the involved ser-

vice instances. The recovery of failed service instances should feature the same invocation data at any substitution service instance. An active service instance should be assigned only to one recovery node that handles its failures, whereas the others should not interfere with it. In case of redundant recovery nodes, assignment of service instances among them should be unique and consistent, as well.

A straightforward approach to reliability of distributed workflow execution, in an ideal sense, implies the exploitation of metadata repositories for the replication of workflow instances. Although such an approach is as simple as publishing the workflow instance to a repository (upon a completion of an activity) it does not favor scalability and consistency.

Given a high workload on the system, that is reflected in a high number of concurrently running workflow instances, all succeeding service instances, that are subscribed for this data at the repository, would entail a huge data propagation overhead. Especially, if the repository service is concentrated to a limited number of nodes, the data propagation would constantly be affected by overload situations (e.g., as illustrated in Figure 1.4)). Due to overload at repositories, subscribed service instances would have to wait for workflow instances thus delaying their execution. To cope with overload, repositories could sacrifice consistency of the propagated data causing this way possibly incorrect recovery of service instance failures. For instance, repositories could either propagate data items as they come in without ordering them logically. Moreover, in face of a failure the repository would have to decide on the substitution service instance, thus further increasing its centrality characteristics and worsens its scalability issue. Another important negative aspect to repository fault-tolerance is its uneconomical resource consumption. That is, each workflow instance would have to be replicated to all subscribed service instances, which is in the absence of node failures a significant waste of resources (e.g., storage, bandwidth etc.). On top of all, the repository nodes would always remain single point of failure.

Safety-Ring Concept

To meet the requirements of an ideal distributed workflow management engine to the biggest possible extent, in the context of reliable execution, we introduce the *Safety-Ring* system service:

$$s_{sr} \in S$$

The main idea behind the Safety-Ring service is to support the reliability of the control flow by a set of peers that monitor the migration of workflow instances and recover failures of active service instances, i.e., SI_a, that would otherwise disrupt the control flow. The peers equipped with Safety-Ring service are referred to as the *SR-nodes* and are denoted with the set SI_{sr}:

$$SI_{sr} = \{si_{sr} \in SI | \forall si_{sr} \in SI : si_{sr}.s = s_{sr}\}$$

Essentially, the SR-nodes perform monitoring of active service instances for each running workflow instance. If necessary, SR-nodes recover failed service instances at a

replacement node of the same service type by invoking the replacement node with the same workflow instance. The assignment of active service instances to SR-nodes w.r.t. monitoring and recovery responsibility is determined by the mapping function f_{mt} as follows:

$$f_{mt} : SI_a \mapsto SI_{sr} \textbf{ with } f_{mt}(si_a) = si_{sr}$$

Thereby, each active service instance necessarily has to be assigned to a SR-nodes if the control flow is to be completely reliable:

$$\forall si_a \in SI_a \exists si_{sr} \in SI_{sr} : f_{mt}(si_a) = si_{sr}$$

The SR-nodes are not dedicated resources of an external system that is assumed to be stable. Rather, any node of our Peer-To-Peer environment can freely participate to the Safety-Ring as well. All it takes is to locally instantiate the Safety-Ring service s_{sr}. Since we operate in heterogeneous environments we cannot assume the SR-nodes to be stable or reliable by default. This implies that even the SR-nodes need to be recovered, in the event of their failure, if the control flow is to be fully reliable. In order not to endlessly delegate the responsibility of SR-node recovery to some higher level or external monitoring authority[1] novel approaches have to be devise. The straightforward yet effective solution of Safety-Ring to this problem is association of SR-nodes among each other for monitoring and recovery purposes.

Given the biggest possible set of SR-nodes, which is N, it is important that association among SR-nodes is achieved efficiently and unanimously such that in the end no SR-node is left unassociated. A failure of a monitoring SR-node should be also immediately and unanimously handled by some other SR-node out of the set of all remaining SR-nodes. Likewise, a new node joining the Safety-Ring should be immediately and unanimously assigned to a SR-node for monitoring and should be assigned with a SR-node which should be monitored by the new SR-node. To this end we need to devise an way of assigning SR-nodes among each other (for monitoring purposes) that can easily incorporate changes in face of SR-node joining and leaving such that the least possible amount of existing SR-nodes is affected. The Safety-Ring approach rules each SR-node to be assigned with exactly one other monitoring SR-node, such that all SR-node monitoring associations are unique. The bijective mapping function f_{sr} thus determines unique assignment among nodes as follows:

$$f_{sr} : SI_{sr} \mapsto SI_{sr} \textbf{ where } \forall si_{sr} \in SI_{sr} \nexists si_x \in SI_{sr} : f_{sr}(si_{sr}) = f_{sr}(si_x)$$

By consequently applying the function f_{sr} on all SR-nodes of SI_{sr} we will obtain a closed chain of monitoring assignments that is of at least length 3. In other words, we will obtain a set of SR-node relations $\prec_{sr}$ that is a transitive closure as follows:

$$\prec_{sr} = \{(si_{sr}, si'_{sr}) | \forall si_{sr} \in SI_{sr} \exists si'_{sr} \in SI_{sr} : f_{sr}(si_{sr}) = si'_{sr}\}$$

The big idea behind the Safety-Ring is to implement the closed chain of monitoring associations (i.e., $\prec_{sr}$) by means of a node ring topology. Thereby, the chosen ring topology should produce a structuring of SR-nodes, such that it semantically corresponds to the

[1] Quis custodiet ipsos custodes.

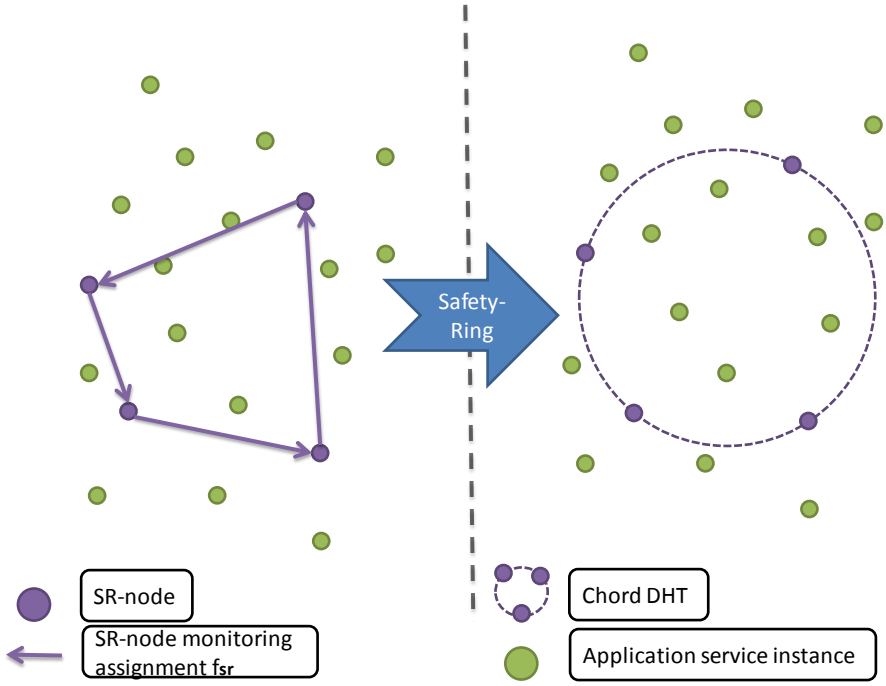


Figure 5.2: Safety-Ring transformation of the SR-nodes into a ring topology.

monitoring associations obtained by means of $\prec_{sr}$. In exploiting the circular identifier space KI of Chord, associated SR-nodes can be placed in a way such that the monitoring SR-node succeeds the other one within KI. This implies that the SR-node monitoring assignment function f_{sr} is implemented in Safety-Ring as the node successor function *succ* of a Chord ring topology:

$$succ(si_{sr}.n) = n' \iff (si_{sr}, (s_{sr}, n')) \in \prec_{sr}$$

Figure 5.2 illustrates the Safety-Ring intuition behind the transformation of SR-nodes into a ring topology.

Therefore, the SR-nodes construct a ring topology of Chord. In doing so, an abundance of inherent benefits is reaped. First of all, the self-organization feature of the ring topology can be exploited so as to efficiently add SR-nodes into system – remove SR-nodes from the system – while leaving the majority of SR-nodes unaffected. Based on Chord only three SR-nodes are affected whenever a new SR-node joins or leaves the environment. In the former case a new SR-node is interjected between two adjacent SR-nodes of the ring topology. In the later case a failed SR-node is handled by connecting the preceding SR-node and the succeeding SR-node of the ring topology. Moreover, scalable means of communication, such as the Finger Tables, among the nodes can be exploited as well. By applying Chord on top of the SR-nodes ring topology scalable data lookup and sharing is possible. This allows for the utilization of reliability mechanisms of Chord so as to enforce reliability of the SR-nodes themselves. For instance, all the data a SR-node locally maintains can be replicated to a set of other SR-nodes by means of symmetric replication.

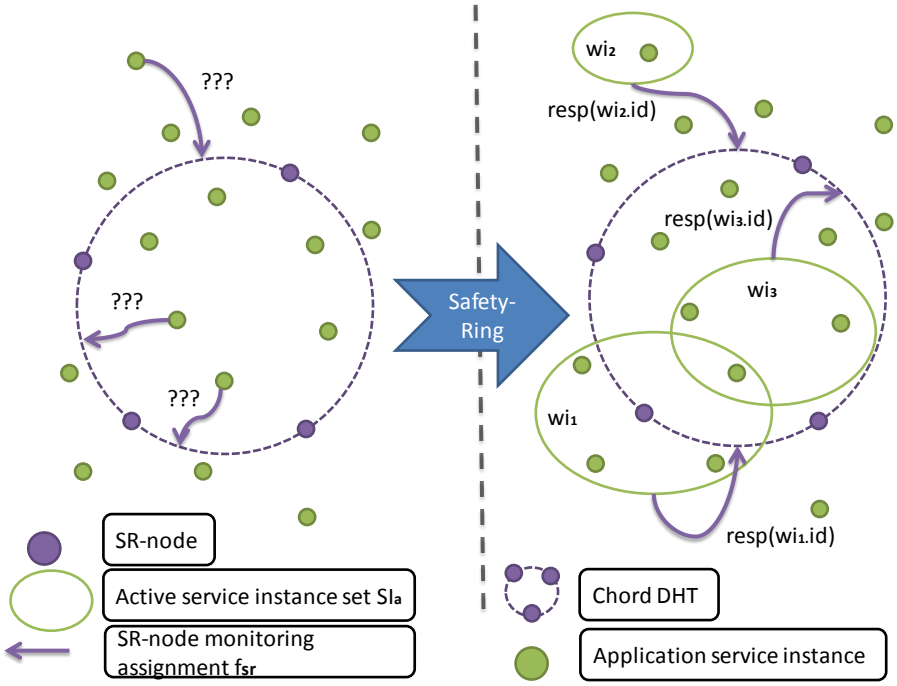


Figure 5.3: Safety-Ring assignment of workflow instance encompassed service instances to the SR-nodes.

The most important benefit of Chord ring construction out of the SR-nodes is however the inherent possibility to efficiently assign active service instances to SR-nodes for monitoring purposes. Given the circular key identifier space KI that is partitioned among the constructing SR-nodes, service instances can be mapped into it as well. In the context of workflow instance execution, active service instances (i.e., in SI_a) are always determined at runtime by late-binding logic of the orchestrator nodes. Thereby, the selection of the concrete active service instances is completely independent of the current workload of the SR-nodes. Note, that SR-node workload is reflected in number of already assigned service instances for monitoring. Mapping the active service instances directly into KI (i.e., by means of f_n) could potentially result in uneven load at the SR-nodes. For example, given a high number of concurrently running workflow instances, a majority of them could bind different service instances that in turn directly map to the same (overladed) SR-node, while the other SR-nodes are underutilized.

Hence, Safety-Ring decouples the SR-node workload distribution from orchestrator late-binding, which is agnostic to SR-node workload. Instead Safety-Ring maps each workflow instances directly into KI by means of its identifier (i.e., $wi.id$) and thus into a key partition of a SR-node. In doing so, the whole active service instance set of an assigned workflow instance (i.e., $wi.AI_s$) becomes subject to monitoring and recovery by the responsible SR-node.

$$\forall wi \in WI \exists si_{sr} \in SI_{sr} : \forall si_a \in wi.SI_a \implies resp(wi.id) = si_{sr}.n \wedge f_{mt}(si_a) = si_{sr}$$

The resulting mapping will yield an efficient and uniform assignment of workflow instances among SR-nodes. Due to mapping of workflow instances into KI based on the

consistent hash function the load distribution is inherently guaranteed to be uniform even in face of high workflow instances numbers. Figure 5.3 illustrates the assignment of workflow instances to SR-nodes.

Observe from Figure 5.3 that application service instances can be overlapping w.r.t. encompassed workflow instances at the same time. Due to late-binding of the control flow, different workflow instances can bind the same service instance in a different context. This implies, that the overlapping service instances can be monitored by multiple SR-nodes at runtime as well.

As an alternative to workload distribution at SR-nodes based on workflow instances is the enrichment of the late-binding logic with SR-node workload metadata. This incurs however additional runtime overhead in mapping the candidate selected service instances to SR-nodes by the orchestrator for decision making. Mapping of service instances into KI by an orchestrator necessitates external network communication (with the Safety-Ring) at runtime and thus significantly reduces the late-binding performance. We conclude this alternative to be less efficient as compared to workflow instance assignment to SR-nodes.

The current state of a SR-node is reflected in the set of all currently assigned workflow instances. To conduct the recovery of failed service instances each SR-node needs to keep a backup of the encompassed workflow instances so that it can invoke a recovery node with the same workflow instance. Moreover, the SR-node state should be shared with other SR-nodes for reliability purposes. Each workflow instance is materialized in the context of the Safety-Ring as a Chord data item that is subject to storage at SR-nodes. For this purpose the mapping functions f_{wd} and f_{dw} are introduced as follows:

$$f_{wd} : WI \mapsto DI \wedge f_{dw} : DI \mapsto WI$$

State sharing among SR-nodes is consequently conducted by means of (symmetric) replication. To enforce consistency of the replicated SR-node state distributed transactions are necessary. Every change to SR-node state, i.e., workflow instance updates, thus needs to be powered by a distributed transaction (e.g., Paxos commit) at the congruent SR-nodes of the corresponding equivalence class.

Therefore, to meet the requirements of an ideal distributed workflow management engine, from a reliability point of view, the Safety-Ring service is essentially tailored to be a synergy of the distributed workflow execution model and a transactional Chord DHT. The Safety-Ring service is formally defined as follows:

Definition 5.1 (Safety-Ring – SRS)**.** *The Safety-Ring is defined as* $SRS \subseteq (DHT_t \times WI)$ *where* DHT_t *transactional Chord DHT and* WI *the set of all workflow instances subject to safeguarding by the Safety-Ring.* □

The aforementioned Definition 5.1 implies that each workflow instance wi is stored by means of a distributed transaction dtx inside a (symmetric replication) Chord DHT such that it stores wi at a node n which is the SR-node responsible for wi:

$$\forall wi \in WI \exists((dht_s, dtx), wi)) \in SRS \implies$$

$$(dht_s.dht.n, s_{sr}) \in SI_{sr} \wedge resp(wi.di) = dht_s.dht.n \wedge f_{wd}(wi) \in dtx.Items$$

Control Flow Support by Safety-Ring

To enhance the control flow execution model for reliability, it has to be slightly altered in extending it with an additional migration step of the workflow instance. This step features the reliable storage of the workflow instance at the Safety-Ring. Being a subject to loss, whenever an activated service instance fails, the workflow instance has to be reliably stored as a backup.

Whenever a service invocation completes a succeeding one is chosen for invocation. The succeeding service instance thus becomes subject to monitoring in place of the finished one. Hence, the control flow based change of the workflow instance state (i.e., change in active activities sets A_a and SI_a) has to be propagated to the Safety-Ring as well. Otherwise the corresponding SR-node would feature an out-of-date state w.r.t. the workflow instance sets A_a and SI_a. Correct recovery based on stale is not possible.

Since the change of workflow instance state is conveyed by means of migration to the succeeding service instance, the control flow has to be redirected towards the Safety-Ring first. Simply put, the late binding of the succeeding service instance has to be presumed with a transactional write of the workflow instance to the Safety-Ring:

$$wi.A_a \xrightarrow{\prec_{cf}} wi.A'_a \implies \exists((dht_s, dx), wi) \in SRS : C \in dx.Items$$

Once the distributed transaction has been completed the workflow instance can be migrated to the succeeding application service instance, in the traditional control flow sense.

In opting for a SR-node workload distribution that is based on workflow instance identifiers we reap another significant benefit. Assuming the workflow instance identifiers (i.e., *wi.id*) to be unique, each distributed transaction will feature only one data item, i.e., the workflow instance itself, for write operations:

$$\forall dtx \in DTX \exists wi \in WI : dtx.Items = \{(f_{wd}(wi), W), C\}$$

As a result the distributed Paxos commit transactions will feature a better performance due to the reduced message overhead. As workflow instance identifiers are unique, transactional writes of concurrent workflow instances will not interfere with each other at the data store of the Safety-Ring and thus reduce the commit conflict probability of the concurrency control mechanism. As the matter of fact, commit conflict potential is reduced to concurrent writes of the same workflow instance. This can only occur when service instances of parallel activity branches migrate the workflow instance at the same time, i.e., write to the Safety-Ring.

Algorithm 5.1 presents the reliable migration of workflow instances that is provided by the Safety-Ring. Observe that in case of the Safety-Ring service type (i.e., Algorithm 5.1, line no.15) the decision is not based on any metadata provided by repository. In principle, any SR-node can be addressed with a workflow instance write request, as the routing of it towards the responsible SR-node will be performed by means of Chord. Hence, we say the service instances interact with the Safety-Ring itself rather than with a specific SR-node.

Only when the updated workflow instance state has been successfully stored to the Safety-Ring, the SR-nodes can correctly recover failures of the active service instances.

Algorithm 5.1 $si_o.migrateSR(wi)$ Invokes activated service instances. Updates afterwards the given workflow instance wi state, in terms of activity sets $wi.A_c, wi.A_a$ and A_s. Writes the updated workflow instance to Safety-Ring.

1: **for** $si \in wi.SI_a : si.n =$ **this do**
2: $si.invoke(wi)$
3: **if** $si.a \in S_d$ **then**
4: $wi.A_a \setminus si.a$
5: $wi.SI_a \setminus si$
6: $wi.A_c \cup si.a$
7: **end if**
8: $A_{next} \subseteq wi.A_s : \forall a_{next} \in A_{next}\ (si.s, a_{next}) \in wi.wf.\ \prec_{cf}$
9: **for** $a_{next} \in A_{next}$ **do**
10: $(si_{next}, rk) \leftarrow n.getSI(a_{next})$
11: $si_o.updateStatistic(a_{next}, rk)$
12: $wi.A_s \setminus a_{next}$
13: $wi.A_a \cup a_{next}$
14: $wi.SI_a \cup si_{next}$
15: $si_{sr} \leftarrow n.getSI(s_{sr})$
16: $si_{sr}.commit(n.getItem(wi))$
17: **end for**
18: **end for**

The stored workflow instance features the latest execution state that can be used by the responsible SR-node to invoke a replacement service instances with the latest workflow instance. Hence, late-binding of succeeding service instances can only occur following a successful Safety-Ring write. For this to happen the failure prone execution model as shown in Figure 5.1 has to undergo slight changes. The reliable workflow instance execution model based on Safety-Ring is illustrated in Figure 5.4.

Example 5.2

Figure 5.4 depicts an example of the reliable execution model for workflow instances. This illustration is based on the example model already introduced in Figure 4.4. However, in Figure 5.4 the SR-nodes I, J and K are additionally introduced to the execution environment. The SR-nodes are depicted with purple colored circular shapes and the monitor icon. Moreover, their interaction with the other nodes (i.e., workflow instance storage at them) is depicted by means of purple arrows. The Safety-Ring nodes are organized into a ring topology, depicted with dashed and blue circle.

As Figure 5.4 shows each service instance has to migrate the workflow instance (i.e., write it) to the Safety-Ring before the subsequent service instance can be activated. On the other hand, once the migration towards the Safety-Ring has been successfully completed, the subsequent service instance invocation is performed by one of the SR-nodes. Moreover, joining of the workflow instances is also performed at the Safety-Ring.

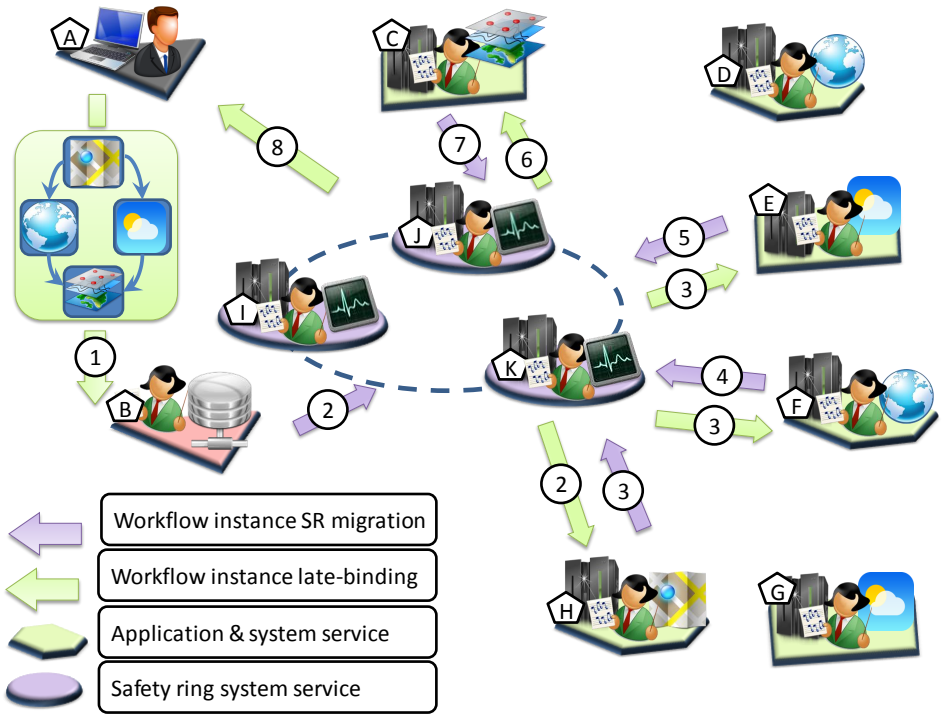


Figure 5.4: Distributed orchestration service execution model with Safety-Ring

As Figure 5.4 suggests, there is more to the Safety-Ring than just plain storage of workflow instances. By being reliable (i.e., consistently replicated), SR-nodes can be entrusted with the responsibility of migrating and joining the control flow in conjunction to monitoring of active service instances.

A workflow instance is not just stored at the SR-nodes for back-up purposes as the result of a transactional write. It is also analyzed w.r.t. its featuring activity sets A_a and SI_a. The storing SR-node of the workflow instance monitors all service instances of the set SI_a at the same time. Once the storing SR-node receives the commit outcome of the transaction (i.e., successfully stores the workflow instance) it triggers the migration of the control flow, in the traditional sense, by late-binding the all service instances of SI_a with the workflow instance itself. In other words, the SR-node migrates the workflow instance instead of the finished service instance.

In late-binding the succeeding service instances the SR-node reviews their activity types w.r.t. potential joins. In case the succeeding activity is not a joining one the SR-node can immediately late-bind the succeeding service instances. On the other hand, if the succeeding activity is a joining one, then the SR-node has to wait till all service instances have written their workflow instances to it before it can proceed. Once all parallel service instances have written their updated workflow instances to the Safety-Ring the corresponding SR-node can merge all updates and trigger the late-bind of the succeeding service instances.

Given the symmetric replication of data items in the Safety-Ring, each workflow instance state update will be shared by a set of SR-nodes. However, only one of them takes an active monitoring and late-binding role, whereas the other ones only serve as

plain data item storage sites, in the traditional DHT sense. Otherwise, active monitoring by the entire workflow instance SR-node sharing set, would entail unnecessary resource consumption (e.g., redundant heart beat messaging) and additional coordination (e.g., in the event of a service instance recovery) overhead among the SR-nodes. To select the active SR-node out of the set of congruent SR-nodes we chose the SR-node that is responsible for the data item with the replication factor enumerator 1 of the congruent equivalence class.

The handling of a written workflow instance at the corresponding SR-node is shown in Algorithm 5.2. This algorithm represents a redefinition of the symmetric replication DHT data item storage routine as shown in Algorithm 3.37. To merge the parallel activity workflow instances into one the routine `mergeWfInstances()` is provided. By default `mergeWfInstances()` performs an union of all parallel workflow instances. By redefining this routine custom merging can be provided as well. Late-binding of the service instances with the workflow instance is conducted by the SR-node by means of the routine `migrateSR()`.

Algorithm 5.2 *n.putItem(item, i)* Aggregates all data items under the same key *ki*. When all workflow instances have arrived merges them. In case this node is replication factor enumerator is 1, ads activated service instances to the fault detector and removes finished service instances from it. Finally, it migrates the workflow instance to the activated service instances.

1: $ki \leftarrow n.getRep(item, i)$
2: $(wi, a) \leftarrow n.getWI(item.di.d)$
3: $JoinWIs[a] \cup wi$
4: $Preds \subseteq wi.wf.A : \forall a_p \in wi.wf.A \exists (a_p, a) \in wi.wf. \prec_{cf}$
5: **if** $|Preds| = |JoinWis[a]|$ **then**
6: $wi_m \leftarrow n.mergeWfInstances(JoinWis[a])$
7: $migrate \leftarrow true$
8: $item_m \leftarrow getItem(wi_m)$
9: **end if**
10: $n.put_s(item_m.di, ki, i)$
11: **if** $i = 1$ **then**
12: $Discrete \subseteq wi_m.SI_a : \forall (s, n) \in Discrete, s \notin S_d$
13: $Monitored \cup Discrete$
14: $Finished \subseteq Monitored : \forall (s, n) \in Finished, s \in wi_m.A_c$
15: $Monitored \setminus Finished$
16: $faultDetector.update(Monitored)$
17: **if** $migrate = true$ **then**
18: **for all** $(s_i, n_i) \in wi_m.SI_a$ **do**
19: $n_i.migrateSR(wi_m)$
20: **end for**
21: **end if**
22: **end if**

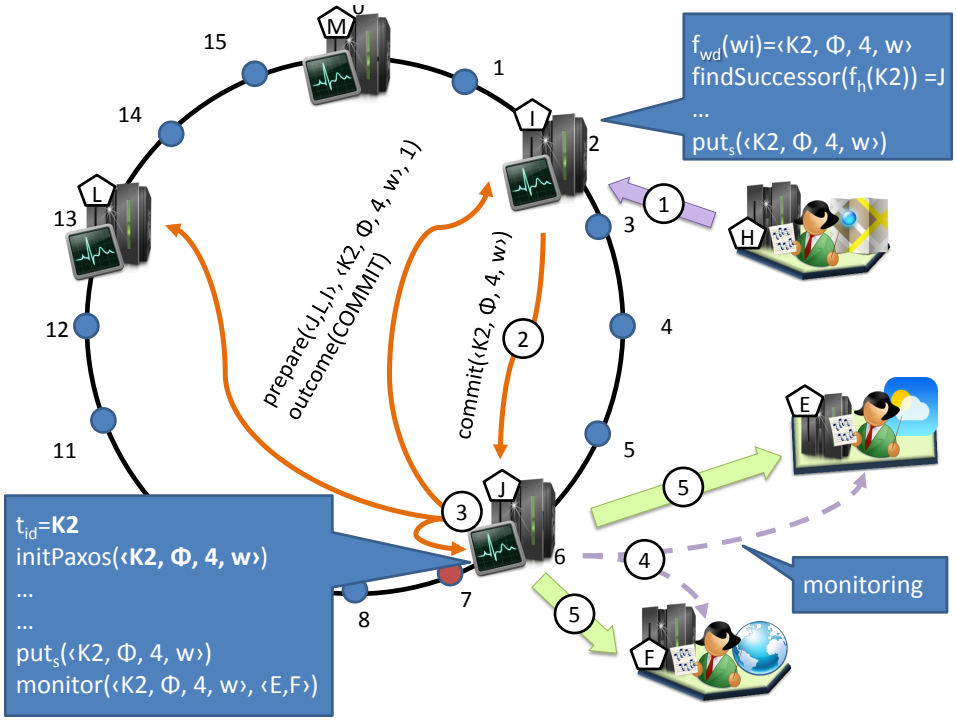


Figure 5.5: Workflow instance late-binding based on the Safety-Ring

Example 5.3

An example of a workflow instance migration enforced by means of Safety-Ring is shown in Figure 5.5. As this example shows the nodes I, J, L and M are organized into a ring topology and are equipped with the Safety-Ring service (denoted with the monitor icon). The positioning of the SR-nodes inside the ring topology implies that I monitors M, M monitors L, L monitors J, and J monitors I per default. As in the examples of Figure 3.4, 5.4 the interactions among nodes are denoted by means of directed and numbered arrows. Thereby, the numbers represent the interaction order, the thin orange arrows internal Safety-Ring node interactions and the thick arrows external interactions with SR-nodes.

In this example the service instance at the node H tries to migrate a workflow instance to the Safety-Ring by addressing the SR-node I (step no.1) first. In turn, the workflow instance is routed as a transformed data item (i.e., $\langle K2, \Phi, 4, w \rangle$) towards the responsible node J at which a Paxos transaction is started (step no.2). Upon successful completion of the workflow instance storage (orange arrows of step no.3) to the SR-nodes I, L and J the data item storage routine put_s is executed. This leads to the migration of the workflow instance towards the service instances at nodes F and E (step no.5), as well as their monitoring (step no.4) by the responsible SR-node J.

Safety-Ring Fault Handling

In case of an active service instance failure, all workflow instances featuring it have to be recovered. To this end the SR-node monitoring the failed service instance has to locate a suitable service instance so as to perform late-binding on it with the backed-up workflow instance. A service instance is considered to be suitable as a replacement if it features the same service type but a different deployment node. To facilitate the lookup of replacement service instances, the fault handling SR-node relies on local metadata stemming from the pub/sub repository. A SR-node creates a subscription at the repository for service instance metadata only when fault handling is performed for the first time. This way metadata propagation is reduced.

Before the actual late-binding of the replacement service instance can take place, the affected workflow instances have to be updated with the replacement service instance, in terms of the set SI_a. In accordance to the reliable control flow model of Safety-Ring, the updated workflow instances have to be written first to the Safety-Ring by the fault handling SR-node. In turn, a successful write to Safety-Ring will trigger the responsible SR-node (i.e., the Algorithm 3.37 will be executed) to perform a late-binding of the replacement service instance. Hence, the distributed control flow will be recovered. To select an optimal replacement service instance out of the set of all service instances the routine `getSI()` is provided. The complete fault handling routine for a failed service instance is shown in Algorithm 5.3.

All service instance failures (both application and system) in the environment are detected by means of an eventually perfect *fault detector* [CGR11] that is applied in the Paxos commit (as elaborated in Section 3.3.2) protocol as well.

Algorithm 5.3 *n.fails(si_f)* Retrieves all affected workflow instances by the failed service instance si_f. Updates them with a substitution service instance and writes the update to the Safety-Ring.

1: $Monitored \setminus si_f$
2: $WIs \leftarrow n.getAffectedWIs(si_f)$
3: $n.updateStatistics(si_f.n, \tau)$
4: **for** $wi_l \in WIs$ **do**
5: $\quad wi_l.SI_a \setminus si_f$
6: $\quad si_r \leftarrow n.getSI(si_f.s)$
7: $\quad wi_l.SI_a \cup si_r$
8: $\quad si_{sr} \leftarrow n.getSI(s_{sr})$
9: $\quad si_{sr}.commit(n.getItem(wi_l))$
10: **end for**

In case of a SR-node failure, the self-healing feature of Chord ring topology is exploited. Recap, in Chord the key partition a departed ring node is automatically taken over by the successor in the ring. In the context of the Safety-Ring, the newly responsible SR-node will obtain the running workflow instances (i.e., by means of Algorithm 3.38) from the other SR-nodes that share the same equivalence class of the key identifier space partition belonging to the failed SR-node. By locally storing the obtained workflow in-

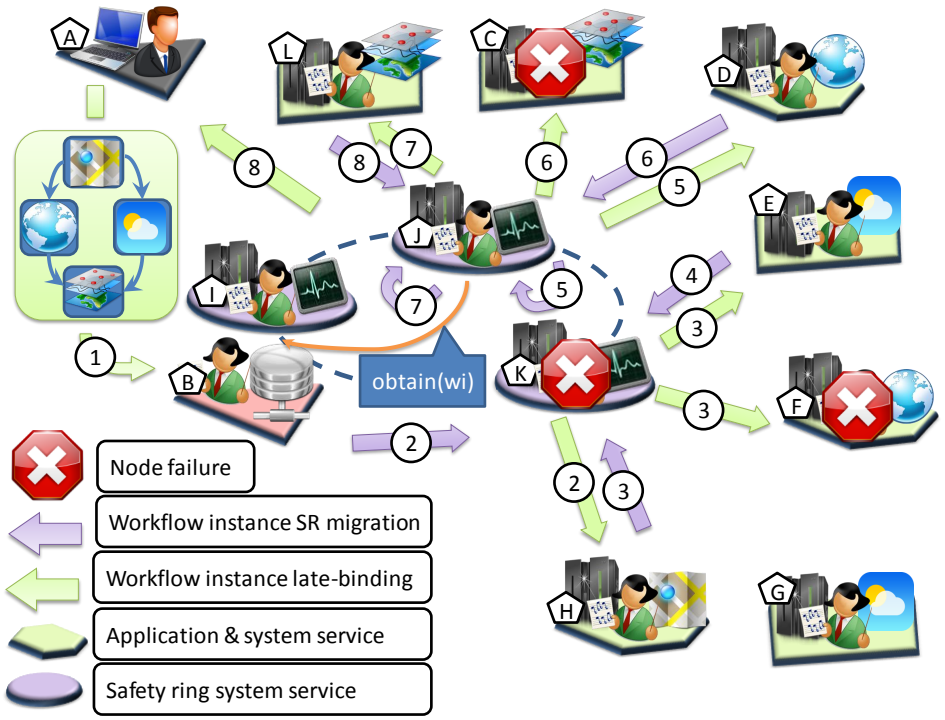


Figure 5.6: Workflow instance late-binding based on the Safety-Ring

stances (i.e., by means of Algorithm 5.2) the fault handling SR-node will resume the monitoring responsibilities of the failed SR-node eventually.

Multiple simultaneous failures of nodes can be handled in the same fashion as well. Each failed application service instance is handled by the responsible SR-node. If the multiple failed service instances are assigned to different SR-nodes they will be recovered in parallel by their corresponding SR-nodes. If multiple failed service instances are assigned to one SR-node, they will be iteratively handled one-by-one by that SR-node.

Multiple simultaneous failures of SR-nodes are handled by the self-organization feature of the ring topology as well. Each successor SR-node will individually recover the failure of its preceding SR-node. In the meantime, the data lookup Finger Tables of the remaining SR-nodes will be inconsistent due to the absence of the failed SR-nodes but will converge eventually once the failed SR-nodes have been handled. However, if multiple and simultaneous failures of SR-nodes occur too frequently the ring topology might split into multiple ones. This is known as the Chord churn problem and has been extensibility addressed in literature [KEAAH05, RGRK04].

Example 5.4

Finally, Figure 5.6 illustrates an example of the reliable execution model provided by Safety-Ring. It builds on the same node environment as introduced in Example 5.2, i.e., Figure 5.4. The node interactions bear the same meaning as well as their equipped services. New to this example are the node failures, illustrated with the red stop icon, and the node L featuring the heat map service, illustrated with the corresponding icon.

Let us assume that SR-node K is responsible for the service instance that enters the system at interaction step 1. Hence, the workflow instance is written to SR-node K in interaction 2 which results in a late-binding of node H. Once node H completes its invocation it writes the workflow instance again to SR-node K which performs a parallel late-binding of the nodes E and F. In after step 3 the node F should fails, only node E will migrate its invocation results to the Safety-Ring at step 4. SR-node K will take notice of the disappearance of node F and initiate the recovery of it. However, if also SR-node K failed before having recovered F the recovery will be performed by some other SR-node. In this example we assume that SR-node J monitors K and thus performs a recovery of it. Since the workflow instance is replicated across the Safety-Ring SR-node J will obtain a copy of the workflow instance from e.g., SR-node I. Once J obtains the workflow instance it will realize that it has to monitor node F only, since in step 4 node E successfully committed its results. Once J notices the disappearance of F as well it will initiate the recovery of it in late-binding node D with the workflow instance. In step 6 node D will write its results to SR-node J where the parallel execution branches of the workflow are merged and forwarded to node C. In case node C fails in the process of late-binding, the workflow instance will be recovered at the node L instead at step 7. Finally, step 8 concludes the execution of the workflow instance.

Safety-Ring Discussion

The Safety-Ring has been built from the bottom up by combining various building blocks (i.e., such as Chord, symmetric replication and Paxos) so as to meet the requirements of an ideal fault-tolerance mechanism. In light of these requirements Safety-Ring fully exploits the benefits of its underlying building blocks in a way which is best summarized as follows:

- Reliability – By delegating the responsibility of late-binding the service instances to the SR-nodes, all possible node failure scenarios of the control flow (e.g., SI_a failures, join failure) are essentially subsumed into two classes of failures:
  - Application service instance failures.
  - SR-node failures.

  Since those failure classes are conceptually overcome again by means of the self-organizing SR-nodes themselves, we conclude the Safety-Ring to be an effective fault-tolerance mechanism for the control flow.
- Efficiency – The Chord based design of the Safety-Ring features efficient self-organization of the ring topology for the purpose of SR-node monitoring assignments, with each SR-node being assigned only one predecessor. This applies even if multiple SR-nodes are frequently joining/leaving the Safety-Ring. In turn, symmetric replication in Safety-Ring facilitates almost immediate restoration of the replication factor in the event of a SR-node failure, i.e., with just one addressed congruent SR-node.

- Scalability – In case, a vast number of workflow instances is run against the Safety-Ring, the load can be accommodated in a scalable fashion by adding new SR-nodes to the ring topology based in a Chord fashion. At the same time uniform load distribution is inherently guaranteed due to consistent hashing of workflow instances in assigning them to SR-nodes.

- Consistency – Symmetric replication based redundancy of workflow instances is always powered by means of distributed transactions (i.e., Paxos commit) for every update of their states. Thereby, consistency is guaranteed even in the case of F SR-node failures out of $r = 2F + 1$ involved SR-nodes.

However, the improved reliability of the control flow by the Safety-Ring does not come without a price. The benefits of Safety-Ring are payed by absorbing the drawbacks of underlying building block.

For instance, the transactional writes towards the Safety-Ring enforce consistent redundancy of the workflow instances, but they also delay their control flow performance. The message intensive transactions of Paxos commit delay the migration throughput of workflow instances at each activity. Each migration step needs to be acknowledged by a transaction before it can proceed to the succeeding service instance. A significant transaction processing load is burdened upon the SR-nodes when a large number of workflow instances (featuring many migration steps) is run against the system. In the meantime, workflow instances have to wait for their migration to the succeeding service instance. The scalable underlying architecture of Safety-Ring (i.e., Chord) helps to mitigate this price in distributing the load to additional SR-nodes in the topology. On the other hand, it additionally increases the ring maintenance overhead per node by consuming additional bandwidth and storage for the larger Finger Tables.

Moreover, Safety-Ring writes are also susceptible for transactional conflicts. Conflicted transactions have to be resolved with aborts and delayed retries, which as a result incurs additional transaction processing load on the SR-nodes and thus workflow instance delay. Join activity migration steps are especially susceptible for transactional conflicts as concurrent service instances may try to write their workflow instances to the Safety-Ring at the same time. Transactional conflicts do not have to occur due to join migrations only. Node failures in acceptors set or the learners set of a Paxos commit transaction can also cause aborts of transactions that have to be compensated. Eventual consistency of the underlying Chord Finger Tables can also cause the transactional processing to fail. For instance, Paxos commit $PREPARE$ requests might end up at learner nodes which do not posses the requested data item. This in turn will result in aborted transactions as long as the Finger Table based routing is inconsistent among the nodes.

Mainly, due to this significant overhead of the Paxos commit building block inside the Safety-Ring concept, it should be limited to control flow reliability only. This implies that SR-nodes should monitor and recover discrete services only as they are the primarily carry out the control flow. It would be theoretically possible to apply the Safety-Ring in the context of streaming services by storing the continuous data flow redundantly at the SR-nodes. The price of such an approach would be however significantly delayed data flow throughput as a result of burdening the inherently resource limited (streaming) nodes with the Paxos overhead. For instance, writing frequently batches

of the continuous data items for multiple streaming instances to the Safety-Ring would certainly overload the SR-nodes and thus affecting the whole data flow. Moreover, It is also very unlikely that the Safety-Ring could successfully store a potentially infinite data flow even if it is continuously scaled out with new SR-nodes. Hence, continuous data flows necessitate more resource friendly approaches to fault-tolerance.

In general, the Safety-Ring reliability mechanism can only work as long as there are enough SR-nodes and application service instances in the execution environment to substitute the failed ones. Moreover, we assume all discrete service types to be fail-safe, i.e., they do not leave behind any uncompensated side effects when they fail. Failures of service types with side effects could be handled by the Safety-Ring mechanism though if the side effects were exposed to the Safety-Ring by storing them inside its data store. Semantic recovery of failed service instances that is characterized by compensations, i.e., a set of semantically equivalent service invocations, is outside the scope of Safety-Ring recovery mechanism as well. Although SR-nodes are capable of late-binding service instance they lack the reasoning about semantic equivalence. For this to work the workflow definitions would have be enhanced with semantical preference orders, and the Safety-Ring nodes would have to be equipped with the reasoning tools so as to deduce and enforce the semantically equivalent alternative paths on the fly.

The aforementioned issues leave room for improvement of the basic Safety-Ring concept as we shall see in the upcoming sections. However, Safety-Ring ensures reliability of the distributed workflow model only at the price of slightly reduced throughput but not at the price of scalability as we shall se in the evaluation chapter of this thesis.

5.1.2 The Safety-Ring Compass Extension

The backbone of the Safety-Ring is the Chord protocol that offers a distributed approach to data management. Essentially Chord avoids any bottleneck and single point of failure while guaranteeing scalable data access with a asymptotic complexity of $O(log(n))$ node traversing steps in a network consisting of at most 2^m nodes, i.e., $n \leq 2^m$. By optimizing the number of node forwarding steps for data access, Chord implicitly assumes that all connections between nodes have comparable bandwidth and latency characteristics. However, in heterogeneous Peer-to-Peer systems that consist of both mobile and static nodes, the Chord approach, despite all of its benefits, is not feasible for its unawareness to network dynamics.

Due to the mobility of some participating nodes, connection parameters can dynamically change over time. For instance, nodes might move away from static ones and thus decrease their data transmission capability or nodes might move closer to static ones and thus increase their data transmission capability. Focusing on a minimal number of node forwarding steps so as to access data in Chord will inevitability entail mobile nodes on the path that decrease the data transmission capability. An example of such a scenario is provided in Figure 5.7.

Example 5.5

Figure 5.7 illustrates a Chord ring that has been already introduced in Figure 3.2, however in this example the ring is composed of heterogeneous nodes. Precisely, the nodes

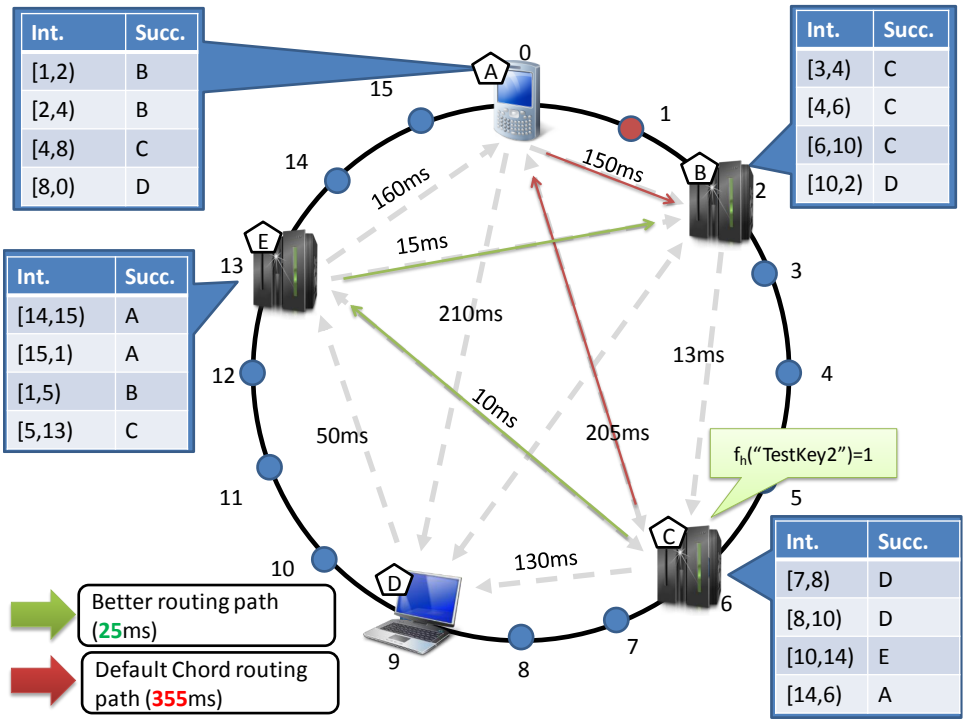


Figure 5.7: The heterogeneous environment Chord problem.

D and *A* have been substituted with less powerful and mobile devices (depicted with the laptop and handy icons). As a consequence, those two nodes will feature reduced network connectivity capabilities. To denote dynamic connectivity among nodes, dashed, gray and directed arrows are used which represent the forwarding directions of the Chord Finger Tables for each node. Moreover, each Finger Table connection is characterized by the current network transmission capability which is represented in our case by means of node round trip latency. As the figure shows the latencies between static nodes are lower (e.g., between node *C* and *E* – 10*ms*) whereas latencies between mobile nodes are higher (e.g., between node *A* and *D* – 210*ms*).

A data lookup for the key identifier value 1 at node *C* would in the default Chord sense (i.e., depicted with the red arrow) require the traversal of node *A*, which is a mobile node. As a result the lookup time would incur a total latency cost of 355*ms*. However, as the figure shows, this is not the best path there exists. If node were to use another Finger Table node instead for forwarding, i.e., node *E*, and node *E* directly node *B* (i.e., depicted with the green arrow), then the total lookup time by summing up the latencies would incur a total cost of 25*ms*. This path is by far more efficient than the default Chord data lookup path. Obviously, even is such a simple example, a mobile node may impact overall latency and the deviation from standard Chord routing can yield significant performance improvements when accessing data.

In the context of Safety-Ring default Chord implies affected transactional processing of workflow instances (at migration time) due to significantly delayed data access paths. As a consequence, the overall system performance, in terms of workflow instance

throughput, should be lowered as well. Hence, efficient data access inside the Safety-Ring, facilitates in the long run better throughput and faster failure recovery. As the example in Figure 5.7 suggests, given a mobile Peer-to-Peer environment, Chord data access should aim at reducing the overall data lookup times, rather than the number of hops in the network.

The Compass Concept

The obvious solution to making Chord feasible in heterogeneous environments involves optimizing the data lookup routine for latency, even if this necessitates additional hops in the network as compared to default lookup routine. *Compass*, a latency-based global optimization algorithm for Chord, addresses this issue. The idea behind Compass is to dynamically derive data access paths that feature a minimal overall latency as a distance metric – which are predominantly composed of powerful (i.e., static) nodes – while preserving the fundamental data access features of Chord such as scalability and consistency.

Since Chord bases its data lookup routine on the key identifier space KI for the purpose of destination identification a latency optimal path has to be based on the same metric as well. However, maintaining latency optimal paths for every key identifier in KI is certainly not scalable and efficient w.r.t. storage. In order not to extend our view of nodes beyond the scalable Finger Table size (i.e., $\log(|KI|)$), Compass selects at each node for any key identifier a local Finger Table node (i.e., fn) that leads to it in the least possible distance w.r.t. total latency. Simply put, Compass assess the finger nodes w.r.t. their capability to reach a key identifier in minimal latency distance. In doing so we complement the hop optimal of Finger Tables with latency information. This implies that in case of missing latency optimal paths towards key identifiers the default Finger Tables can be used so as to advance the data lookup process. Hence, Compass defines a latency optimal path with a destination key identifier, a local finger node that leads to it, and the latency distance metric that is needed to reach it. The union of all optimal forwarding nodes for all key identifiers, of minimal latency, constitutes the *Compass Table*. Formally, the Compass Table is defined as follows:

Definition 5.2 (Compass Table – *ct*)**.** *A Compass Table entry at a node $n \in N$ is defined as tuple $ct(n) = (n_{fw}, KI_{ct}, lt)$ where:*

- *$n_{fw} \in FT(n)$ is optimal forwarding finger node,*
- *$KI_{ct} \subseteq KI$ is set of destination key identifiers for which n_{fw} is optimal,*
- *$lt \in \mathbb{R}^+$ is approximated latency towards the destination key identifiers KI_{ct} from node n*

The set of all Compass Table entries for node n is defined as $CT(n)$, whereas the set of all Compass Table entries for nodes is defined as CT. □

Note, that multiple key identifiers are likely to feature the same optimal forwarding node and the same latency distance at runtime, hence they can be subsumed into one Compass Table entry featuring the unified destination key identifier set, i.e., KI_{ct}. Naturally, the key identifier partitions that are assigned to the directly connected finger

nodes can be immediately derived by means of periodical latency probing. To associate two nodes w.r.t. their relative and directly probed latency the function f_{lat} is introduced as follows:

$$f_{lat} : N \times N \mapsto I\!R^{+} \cup 0$$

The joint key identifier partitions of all finger nodes will however only cover a portion of KI:

$$KIP_n = \bigcup_{\forall fn \in FT(n)} KIP(fn) \textbf{ where } KIP_n \subset KI$$

The rest of the key identifier space (i.e., $KI \setminus KIP_n$) is not directly reachable, but it can be reached via a series of hops beginning from one of the finger nodes. To select the optimal forwarding finger node, that features the smallest latency towards a destination key identifier, the distance on the whole path to the remote destination key identifier has to be approximated. To this end Compass exploits greedy shortest paths algorithms, in particular the *Bellman-Ford* [Bel58] algorithm on top of the Compass Table.

Optimal Path Computation

The Bellman-Ford algorithm initially associates to a path a maximal distance value (∞), which is then based on the relaxation principle, gradually minimized until the optimal value is computed. Relaxation of the current distance value to a key identifier is performed in iterations, by greedily assessing all finger nodes for the distance towards the destination key identifier for each iteration. Consequently, the finger node that features the smallest distance towards the destination key identifier is greedily chosen to be the new forwarding node n_{fw} and its destination value is taken over. The Bellman-Ford relaxation process on top of Compass Tables can be summarized by the following equation:

$$\begin{gathered} \forall ct(n) \in CT(n) \\ ct(n).lt = \arg \min_{fn \in FT(n)} \{ct(fn).lt + f_{lat}(n, fn)\} \wedge ct(n).n_{fw} = fn : \\ ct(fn) \in CT(fn) \wedge ct(n).KI_{ct} = ct(fn).KI_{ct} \end{gathered} \tag{5.1}$$

For the relaxation process to result in latency minimal paths for each key identifier the Bellman-Ford algorithm mandates that all finger nodes have to be assessed $|N|$ times, i.e., for each node in the environment one iteration is necessary. Since each Compass node is only aware of the directly connected finger nodes it has to retrieve knowledge on the other nodes implicitly. To this end each Compass node periodically obtains the Compass Tables of its finger nodes. Likewise, each node forwards its Compass Table to all other nodes for which it is a finger node. The information inside the Compass Tables, stemming from the finger nodes, is exploited for the relaxation of the latency paths, in updating the forwarding nodes and latencies, as shown in Equation 5.1. Hence, Compass applies a gossip based protocol for Compass Table retrieval among the nodes. Given the location of finger nodes in the Finger Table the size of is of logarithmic w.r.t. the key identifier space if the size of KI is an exponent of 2:

$$|FT(n)| = log_2(|KI|) = m \textbf{ with } |KI| = 2^m$$

Then it also takes at most m iterations of the gossip protocol to cover the whole KI by combining all destination key identifier sets KI_{ct} and propagating the Compass Tables among all nodes. This implies that the asymptotic complexity of Compass, in terms of the number of iterations to obtain all nodes, is logarithmic as well:

$$Compass\ convergence = O(log(|KI|)) \approx O(m) \textbf{ where } |KI| = 2^m$$

To start the optimal path relaxation process each node needs to initialize its Compass Table, such that it reflects the node's current knowledge on optimal paths. Initially, the latency distance towards a node's key identifier space partition is clearly 0 whereas the latency distance towards the rest of KI is unknown. To represent an unknown view of KI, a node adds an entry (i.e., $ct_{uknw}(n)$) to the Compass Table that features a *null* forwarding node and a latency distance of value ∞ as follows:

$$ct_{uknw}(n) = (null, KI \setminus KIP(n), \infty)$$

For parts of KI that are featured by the unknown Compass Table entry default Chord routing is used as a fall-back. In case a node is alone in the ring than the latency distance for all of KI is 0 until another node joins the ring. The Compass Table initialization routine is shown in Algorithm 5.4.

Algorithm 5.4 *n.initCompass()* creates a Compass Table entry for node n as itself being the forwarding node with a latency of 0. If n is preceded by *prdecessor*, the encompassing key identifier space KI_{ct} is set to be the key identifier space partition of n, i.e., $KIP(n)$. Moreover, a unknown Compass Table entry is created. In case n is alone KI_{ct} is set to be the whole KI.

1: **if** $predecessor \neq$ **null then**
2: $\quad KI_{ct} \leftarrow KIP(n)$
3: $\quad KI_{rest} \leftarrow KI \setminus KI_{ct}$
4: $\quad CT(n) \cup (n,$ **null** $, KI_{rest}, \infty)$
5: **else**
6: $\quad KI_{ct} \leftarrow KI$
7: **end if**
8: $CT(n) \cup (n, n, KI_{ct}, 0)$

The fully initialized Compass Table is further iteratively refined by means of the Bellman-Ford algorithm (i.e., Equation 5.1) based on the Compass Tables of the finger nodes. In doing so, each node is likely to feature in the course of the relaxation process a different view on KI w.r.t. key identifier sets encompassed by Compass Table entries. Two entries of different nodes might feature key identifier sets which not the same (i.e., are differently fragmented) but are partially overlapping. In order to be comparable for latency distance overlapping key identifier sets have to realigned. Thereby, realignment is achieved by partitioning the local Compass Table entries into new ones encompassing the overlapping key identifier set only. That is, whenever a local Compass Table entry is overlapping with a fn Compass Table entry for an encompassed key identifier set KI_{ct}, locally a new entry is created that features the overlapping key identifier set, thus

reducing KI_{ct} of the already existing local Compass Table entry by the overlapping part. The newly created Compass Table entry additionally features the smaller latency value by means of relaxation of the two comparing entries along with the optimal forwarding node.

Example 5.6

For instance, the Compass Table entry of node C, i.e., $ct_C = (c, A, [13, 2], 205)$, and entry of node E, i.e., $ct_E = (e, B, [1, 2], 15)$ are at a certain point in time overlapping for KI_{ct}. In this example the encompassed key identifier sets of C and E are denoted by means of closed intervals encompassing all key identifiers that are located within the bounds. Precisely, the overlapping key identifier set is $[1, 2]$, i.e., $ct_C.KI_{ct} \cap ct_E.KI_{ct} = [13, 2] \cap [1, 2] = [1, 2]$. In order to realign these entries, the entry ct_C is partitioned into two new ones ct_{C1} and ct_{C2} such that ct_{C2} can be compared with ct_E. Thereby, ct_{C1} denotes the old entry reduced by the overlapping interval (i.e., $ct_{C1} = (c, A, [13, 0], 205)$), whereas ct_{C2} features the overlapped interval (i.e., $ct_{C2} = (c, A, [1, 2], 205)$). Now that the entries ct_{C2} and ct_E are comparable they can be assessed with the relaxation process. Since node E features a much lower latency distance towards $[1, 2]$ than the current value of ct_{C3}, also by accounting for the direct latency towards E (i.e., $ct_E.lt + directLatency_E < ct_{C2}.lt \approx 15 + 10 < 205$), the latency value has to be relaxed to the lower value and the forwarding node to set to be E, i.e., $ct_{C2} = (c, E, [1, 2], 25)$.

The relative latency values among a node and its direct finger nodes are subject to constant change, i.e., they can decrease but they can also increase over time. These changes have to be passed on to the Compass Tables if the computation of optimal paths is to be continuous. For instance, if the direct latency towards one finger node increases then some other finger node is maybe more suitable as an optimal forwarding node. Precisely, the relaxation process converges after m iterations resulting in optimal paths for each key identifier. In case the increase of direct latency is not passed on, the converged paths will never change although they might be out-of-data and suboptimal. That is why the relaxation process of Compass has to be periodically repeated. In doing so, the updated latency values, obtained by the results of the function f_{lat}, will be propagated to the Compass Table entries according to Equation 5.1.

Algorithm 5.5 shows the periodic maintenance process of Compass Tables that incorporates the Equations 5.1. Note, that once the iterative relaxation process of Compass tables has converged, the amount of their entries will always match the number of nodes in Chord:

$$\forall n \in N \implies |CT(n)| = |N|$$

Moreover, all Compass Table entries will feature the same encompassed key identifier sets. This is due to the fact that each Chord node features a unique key identifier space partition that cannot be further partitioned within the scope of the relaxation process:

$$\forall n \in N \nexists n_x : KIP(n) \subseteq KIP(n_x)$$

The unknown Compass Table entry (i.e., $ct(n)_{uknw}$), which represents the rest KI, will be gradually partitioned into smaller pieces, in the course of the relaxation, by chopping off

Algorithm 5.5 *n.maintainCompass()* Periodically iterates at node n over all finger nodes fn and obtains their Compass Table entries. Checks for overlapping key identifier sets $CT_{overlap}$ w.r.t. with fn Compass Tables $CT(fn)$. In case a fn is already the forwarding node n_{fw} for an overlapping key identifier set $KI_{overlap}$ then the latency value is accounted for with the directly measured latency lt_{fn} of fn. In case a fn features a lower latency for $KI_{overlap}$, also wile accounting for lt_{fn}, it selects fn as the new optimal node with the new latency value. For any update of an existing Compass Tables entry, it creates a new entry which is added to the set *Temp*. Finally, it joins all newly created entries of *Temps* that feature the same n_{fw} and lt into one entry w.r.t. to KI_{ct} and adds it to the local Compass Table.

1: **for all** $fn \in FT(n)$ **do**
2: $lt_{fn} \leftarrow fn.getProbedLatency()$
3: **for all** $ct \in CT(fn)$ **do**
4: $Temp \leftarrow \emptyset$
5: $CT_{overlap} \subseteq CT(n) | \forall ct_o \in CT_{overlap} \implies ct_o.KI_{ct} \cap ct.KI_{ct} \neq \emptyset$
6: **for all** $ct_o \in CT_{overlap}$ **do**
7: $KI_{overlap} \leftarrow ct_o.KI_{ct} \cap ct.KI_{ct}$
8: **if** $ct_o.n_{fw} = fn$ **then**
9: **if** $ct.n_{fw} = n$ **then**
10: $Temp \cup (n,$ **null** $, KI_{overlap}, \infty)$
11: $ct_o.KI_{ct} \setminus KI_{overlap}$
12: **else**
13: $Temp \cup (n, fn, KI_{overlap}, lt_{fn} + ct.lt)$
14: $ct_o.KI_{ct} \setminus KI_{overlap}$
15: **end if**
16: **else if** $ct_o.lt > lt_{fn} + ct.lt \wedge ct.n_{fw} \neq n$ **then**
17: $Temp \cup (n, fn, KI_{overlap}, lt_{fn} + ct.lt)$
18: $ct_o.KI_{ct} \setminus KI_{overlap}$
19: **end if**
20: **if** $ct_o.KI_{ct} = \emptyset$ **then**
21: $CT(n) \setminus ct_o$
22: **end if**
23: **end for**
24: **for all** $ct_t \in Temp$ **do**
25: $CT_s \subseteq Temp | \forall ct_s \in CT_s \; ct_t.lt = ct_s.lt \vee ct_t.n_{fw} = ct_s.n_{fw}$
26: $Temp \setminus CT_s$
27: **for all** $ct_s \in CT_s$ **do**
28: $ct_t.KI_{ct} \cup ct_s.KI_{ct}$
29: **end for**
30: $CT(n) \cup ct_t$
31: **end for**
32: **end for**
33: **end for**
34: $n.compressTable()$

the atomic units from it, i.e., the key identifier space partitions of each individual node. Thus after m iterations we will obtain $|N|$ atomic partitions of KI for each Compass Table.

Compass Data Lookup

Once the Compass Tables have fully converged they can be used for latency optimal data access. Given a destination key identifier at data lookup time, the Compass Table is queried for the entry that encompasses it. In case the optimal forwarding node is the queried node itself, the search is concluded, by declaring the queried node the destination. Otherwise the data lookup query is forwarded to the optimal forwarding node until the destination node is reached. In case a Compass Table does not contain information as on where to forward the lookup for a given destination identifier, Compass consults the Finger Tables instead. In such a situation the forwarding node is determined on the basis of the Finger Table hop optimized forwarding. This way the forwarding process remains uninterrupted. Hence, Compass and Chord are jointly responsible for the data access routine and do not mutually exclude each other, rather Compass extends Chord's data access routine for latency. Note, that unknown optimal forwarding node information in Compass Tables are only possible immediately after a table reinitialization, that is while the optimal path computation is still in the process of convergence. Algorithm 5.6 shows the reimplemented data lookup routine so as to feature Compass as well.

The data access paths derived from the Compass Tables are expected to deviate from the Finger Tables paths due to the completely different optimization grounds goals (i.e., latency vs. number of hops). In order for both tables to be nevertheless consistent, in terms of destination node, an important precautionary measure has to be taken by Compass. While Chord based data lookup routine stops at the predecessor of destination node, Compass destination resolution has to take place directly at the destination node itself.

Since Chord is designed to optimize for the number of hops it can afford to stop a lookup at the predecessor. Compass on the other hand, bases its forwarding decision on a metric that is constantly changing (i.e., latency) and cannot be certain about a destination but at destination node itself. Hence, Compass lookup has to always make the extra forwarding step and stop at the destination node only. Hence, consistency between Chord and Compass relies on the on correctness of successor entries at each node for Chord, whereas in Compass consistency relies on the correctness of predecessor node entries. In other words, lookup consistency is determined by the correct predecessor-successor values for each node in the ring topology. Since these values are set by the periodic `stabilize()` (i.e., Algorithm 3.4) routine Compass can be out of sync with Chord for at most as a `stabilize()` invocation period lasts. Therefore, consistency of the lookup routines can be improved by increasing the frequency of `stabilize()` routine invocations. This implies for Chord lookup consistency in general. An example of a Compass based data lookup is illustrated in Figure 5.8.

Algorithm 5.6 *n.findSuccessor(id)* Looks at node n's Compass Table for an entry that encompasses the given key identifier *id* if Compass is enabled and n has a predecessor. In case an entry exists and the forwarding node n_{fw} is the same as n then it concludes the search by returning n. If the entry exists and n_{fw} is not the same as n then it forwards the search to n_{fw}. For all other cases the Finger Table is consulted by calling its $findSuccessor(id)$ routine.

```
1: if compassEnabled = true then
2:     HopHistory ∪ n
3:     if CT(n) ≠ ∅ ∧ predecessor ≠ null then
4:         ∃ct ∈ CT(n)|id ∈ ct.KI_ct
5:         if ct.n_fw ≠ null then
6:             if ct.n_fw = n then
7:                 return n
8:             else
9:                 if ct.n_fw ∉ HopHistory then
10:                    return ct.n_fw.findSuccessor(id)
11:                else
12:                    ct.lt ← ∞
13:                    ct.n_fw ← null
14:                end if
15:            end if
16:        end if
17:    end if
18: end if
19: return FT(n).findSuccessor(id)
```

Example 5.7

Figure 5.8 bases the Compass node environment on the Chord ring as already introduced in the example of Figure 3.2. In this example however, the Finger Tables have been replaced with Compass Tables instead. As the figure shows the all nodes feature completely converged Compass Tables that feature the same encompassed key identifier sets but different latencies w.r.t. their corresponding Finger Table nodes. For instance, the Compass Table of node C features five entries, each entry encompassing a key identifier set that corresponds to one partition of one node in the ring, i.e., $ct_{C1} = (c, E, [1,2], 25)$, $ct_{C2} = (c, C, [3,6], 0)$, $ct_{C3} = (c, D, [7,9], 130)$, $ct_{C4} = (c, E, [10,13], 10)$ and $ct_{C5} = (c, E, [14,0], 170)$. As depicted node E is the only static node out of all Finger Table nodes (i.e., D, E and A) for C, thus it is selected as the optimal forwarding node n_{fw} for most of the entries (i.e.,, ct_{C1}, ct_{C4} and ct_{C5}).

The same data lookup for the key identifier value 1 at node C as in example Figure 5.7 not result in the red path but rather in the green path. That is, consultation with the Compass Table would yield ct_{C1} as the encompassing entry and thus node E as the forwarding node. At node E the same lookup for 1 would yield the entry $ct_{E1} = (e, b, [1,2], 15)$ and node B as the forwarding node. Finally, at node B the lookup would be concluded as $id = 1$ as the Compass Table would return an entry that features

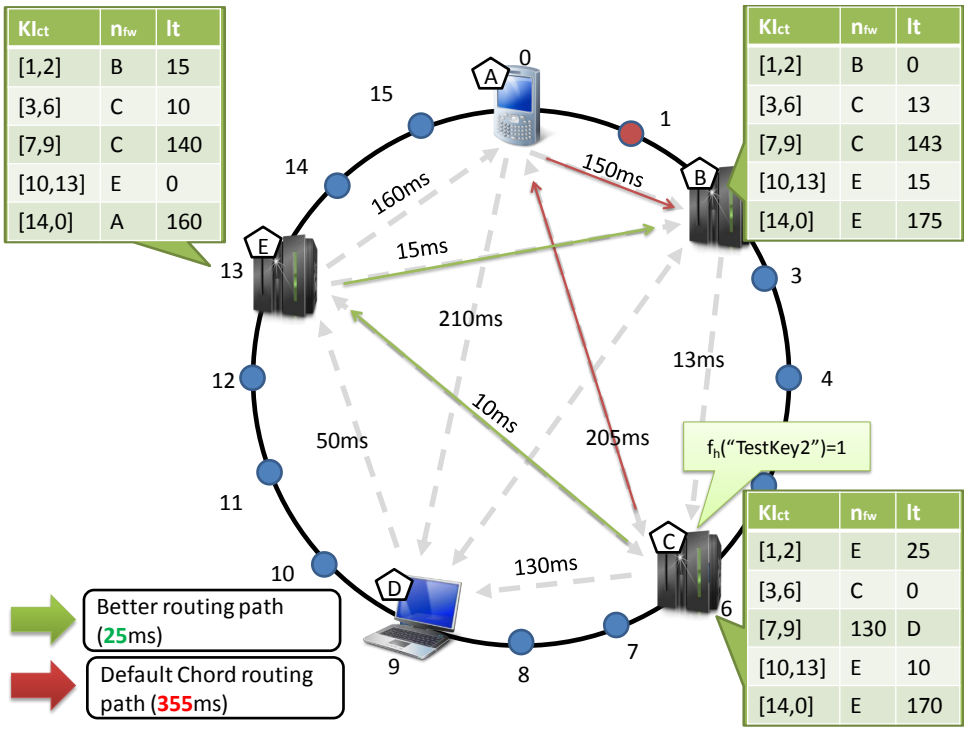


Figure 5.8: Compass routing

the node itself as the optimal forwarding node, $ct_{B1} = (b, b, [1,2], 0)$. By exploiting the Compass Table the lookup would incur latency costs of about $25ms$ in total which is by far more efficient than the default red path.

Compass Dynamics

The latency distance metric on which the Compass is founded is measured value and thus subject to dynamic changes at runtime. Due to propagation of these changes to the Compass Tables the optimal forwarding nodes value tends to constantly change in the process of relaxation. The biggest consequence of frequent (i.e., volatile) optimal forwarding nodes changes is the prolonged convergence and/or even the divergence of the relaxation process itself. Although prolongation of relaxation will not obstruct its goal in minimizing the latency it may however have significant effects on its outcome by devising paths which are far from optimal from a global point of view.

The most common issue to Bellman-Ford based data lookup algorithms, that are caused by volatile latency oscillations, lies in the *count-to-infinity* problem. In the count-to-infinity problem a certain number of forwarding nodes from in the course of a volatile relaxation process an optimal path that corresponds to a loop:

$$\exists ct(n) \in CT(n) \wedge \exists ct(n_x) \in CT(n_x) :$$

$$ct(n).KI_{ec} = ct(n_x).KI_{ec} \wedge ct(n).n_{fw} = n_x \wedge ct(n_x).n_{fw} = n$$

Once a lookup enters this loop it undefinability advances its lookup request only among the nodes forming the loop and thus never reaches its destination.

The count-to-infinity loops are formed when due to a sudden change in direct latency some node reevaluates a new fn to be optimal. At the new optimal node the latency burst affected node is already seen as optimal, for the same key identifier set, thus the loop is formed. In a loop all participants increase the distance metric in a spiral fashion w.r.t. Equation 5.1 by increasing the direct latency to each other until infinity. In theory the count-to-infinity problem should resolve itself after some time, i.e., whenever the distance metric has been increased so much such that another $forwardingNode$ becomes optimal (i.e., in case there is one). However, in practice the resolution of circular paths tends to last be rather lengthy, which in the meantime nullifies the optimal data access paths of Compass. Moreover, infinite lookup looping at such nodes results in an unnecessary waste of local computational and communication resources likely needed for other system functionalities.

To overcome the count-to-infinity problem a number of proactive and reactive measures can be taken. Compass applies latency flattening in conjunction to Poison Reverse. These two solutions can be described as follows:

- *Latency flattening* – First, volatile oscillations of latency measurements, which tend to frequently occur in heterogeneous networks, can be flattened out in a proactive fashion. To this end Compass follows the suggestions presented in literature [RGRK04] by applying an exponentially weighted moving average value on the latency measurements. The following equation shows the oscillation smoothening of latency measurements at each periodic probe:

$$f_{lat}(n_1, n_2)^{new} = \alpha \cdot probe(n_1, n_2) + (1 - \alpha) \cdot f_{lat}(n_1, n_2)^{old} \tag{5.2}$$

  Note, that function *probe(n_1,n_2)* returns the measured relative latency among the specified nodes.

- *Split Horizon with Poison Reverse* – Compass also applies proactive *Split Horizon with Poison Reverse* [Bla00] techniques so as to prevent circular path formation. Whenever a loop is suspected in the process in relaxation, preemptively the suspected Compass Table entry is reset to unknown. This way the unknown value is propagated rather than a potential loop path to the other nodes. For instance, Algorithm 5.5 features Poison Reverse constructs (i.e., lines $9 - 11$) in setting the optimal forwarding node.

- *Reactive Poison Reverse* – Poison Reverse in conjunction to filtered latency measurements facilitate the count-to-infinity prevention for simple loop path situations that involve only two succeeding nodes. To dissolve more complex circular paths (e.g., of three or more involved nodes), which are however rather seldom in practice, reactive Poison Reverse techniques have to be applied. Reactive count-to-infinity prevention approaches focus on detecting the loops while traversing the nodes at data lookup time. In case a loop is detected, the loop closing Compass Table entry is reset to unknown and thus Poison Reverse is applied. Loop detection is made possible by maintaining a the lookup history state. Thereby, the

state can be represented by means of a hop counter or a list of traversed nodes. In Compass, a node hop history is maintained and exploited by the data lookup routine `findSuccessor(id)` (i.e., Algorithm 5.6, lines 1 and $11-14$). That is, in calling `findSuccessor(id)` each traversed node is added to the hop history and in case the optimal forwarding node to be queried for data is already contained in the hop history the loop is detected and Poison Reverse is applied.

Finally, dynamics in the Chord ring topology also have a significant influence on the Compass Tables as well. Inconsistency with the ring topology, reflected in the Finger Tables and predecessor nodes, inevitably results in inconsistency of Compass. Since the Finger Tables are the foundation for Compass changes to them, in face of joining/leaving finger nodes, have to be propagated to Compass Table level immediately. A straightforward, yet effective solution to Finger Table dynamics, lies in case of Compass with the reinitialization of Compass Tables. Simply put, whenever an underlying Finger Table changes, the Compass Table has to be reset so as to restart the relaxation process and thus reevaluate the updated finger nodes. The reinitialization of the Compass Table has to take place whenever the predecessor node changes as well.

Compass Efficiency

The iterative relaxation process of Compass, once converged, results in Compass Tables that are of linear size w.r.t. the number of nodes in the Chord ring:

$$|CT(n)| = |N|$$

The computational complexity of the Compass algorithm can be broken down into the asymptotic complexity of the Bellman-Ford algorithm on which it is based. By definition, the complexity of the Bellman-Ford is bound to the number of nodes (i.e., $|N|$) in the environment and the number of edges (i.e., $|E|$) among them:

$$Bellman-Ford\ computation = O(|N| \cdot |E|)$$

When applied to Compass, each node possess only one edge for each of its finger nodes, i.e., only ($|FT(n)| = log(|KI|)$) edges. However, to obtain all $|N|$ entries of the Compass Table $log(|KI|)$ iterations of the Compass maintenance algorithm are necessary at each node during which all the finger nodes are evaluated all over for optimal paths. Hence, we conclude the asymptotic complexity of computing the optimal paths of the Compass maintenance algorithm to be:

$$Compass\ computation = O(|N| \cdot log(|KI|)^2) = O(|N| \cdot m^2)\ \textbf{where}\ |KI| = 2^m$$

For most computational nodes of a heterogeneous environment such a complexity should prove to be affordable. First, given the our motivation application scenario of Section 2 it is unlikely to expect the to execution environment to be of unlimited node numbers. Rather a predefined and limited setting of computational nodes, as in the case of the mobile computers at the firefighters, stationary computers at the trucks etc. will be used instead. Second, given the advancements in computational performance

(i.e., CPU power and main memory capacity) of modern mobile devices [Gao] the local maintenance of a data structure that is of a quadratic size over a logarithmic base should not be too much of a computational overhead.

In case Compass is applied to a scalable execution environment that is composed of a vast number of computational nodes (i.e., $|KI| \approx |N|$), a linear Compass Table might prove to be inefficient from a bandwidth resource consumption point of view. In particular mobile (i.e., resource-limited) nodes might be significantly burdened with the overhead of Compass maintenance algorithm (i.e., the gossiping part of it) in periodically transmitting the linear Compass Tables to the finger nodes. To reduce the bandwidth consumption at mobile nodes, Compass applies a compression algorithm on top its tables just before they are sent out into the network and thus significantly contributes to network transmission reduction per node.

The compression algorithm is based on the idea of exploiting the destination metric of Compass Tables, which is KI, to create new entries which encompass a bigger partition of KI and thus subsumes multiple affected entries so as to reduce reciprocally the size of Compass Tables. In order to be able to subsume multiple entries into a bigger one they have to be *similar*. Similarity of entries is derived on their relative latency distances towards adjacent key identifier sets for the same optimal forwarding node. Simply put, if two Compass Table entries feature adjacent key identifier sets, the same forwarding node and almost the same (similar) latency distances then they can be merged into one entry. In turn, the newly joined entry will feature a merged key identifier set, the carried over optimal forwarding node and one latency distance value of the two similar values. In order not to overestimate similarity with other entries in the future we choose the higher latency value of the joining entries for the new entry.

Latency similarity, is defined by means of a threshold value σ. This values specifies the relative latency distance among any two Compass Table entries that has to be matched. To compute the relative distance among any two Compass Table entries of a node the distance function f_{sj} is introduced as follows:

$$f_{sj} : CT(n) \times CT(n) \mapsto I\!R^{+} \textbf{ where } f_{sj}(ct_1(n), ct_2(n)) = \frac{|ct_1(n).lt - ct_2(n).lt|}{max\{ct_1(n).lt, ct_2(n).lt\}}$$

Hence, in case two Compass Table entries feature latencies whose relative difference is lower than the latency threshold value σ they have the potential to be merged into one entry. Adjacency of key identifier spaces is determined by means of the function f_{adj} as follows:

$$f_{adj} : 2^{KI} \times 2^{KI} \mapsto \{true, false\}$$

Example 5.8

For instance, let us assume that the latency value of node A improves to $20ms$ for node C and $5ms$ for node E in the example presented in Figure 5.8. As a consequence the entry encompassing the key identifier space partition of node A in node C's Compass Table would change to $ct_{C5} = (c, E, [14, 0], 15)$ eventually. Given the similar entry $ct_{C4} = (c, E, [10, 13], 10)$ that features the same forwarding node E, a subsequent key identifier set ($f_{adj}([10, 13], [14, 0]) =$ **true**) and a similar latency (i.e., $f_{sj}(15, 10) < 0.4$ for $\sigma = 0.4$)

to ct_{C5} they can be merged into a new entry. Thereby, ct_{C5} would be added to ct_{C4} such that the updated ct_{C4} entry would extend its key identifier space set and assume the higher latency as follows: $ct_{C4} = (c, E, [10, 0], 15)$.

The complete Compass Table compression routine is shown in Algorithm 5.7. Note, that Algorithm 5.7 is also periodically applied as the last step of the Compass Table maintenance routine (i.e., Algorithm 5.5, line no.19).

Algorithm 5.7 *n.compressTable()* In case Compass Table compression is enabled at node n, first it selects fer each all entry a subset of the Compass Table that is similar CT_j, in terms of adjacent encompassed key identifier sets KI_{ct}, the same forwarding nodes n_{fw} and lower relative latency distance than σ. Finally, it merges all entries of CT_j into a single one by taking over the maximal latency value of all entries.

1: **if** $compression =$ **true then**
2: **for all** $ct_x \in CT(n)$ **do**
3: $CT_j \subseteq CT(n) | \forall ct_j \in CT_j \implies f_{sj}(ct_x, ct_j) < \sigma \wedge ct_x.n_{fw} = ct_j.n_{fw} \wedge f_{adj}(ct_x.KI_{ct}, ct_j.KI_{ct}) =$ **true**
4: $CT(n) \setminus CT_j$
5: **for all** $ct_j \in CT_j$ **do**
6: $ct_x.lt \leftarrow max\{ct_x.lt, ct_j\}$
7: **end for**
8: **end for**
9: **end if**

The similarity threshold value σ is subject to specification by an expert end-user that administers the workflow engine so as to reflect demands of the high-level application for bandwidth resource conservation. However, σ also introduces a trade-off between precision and efficiency of the Compass approach. By adjusting for σ the compression granularity of the Compass Tables can be specified directly. A higher σ value should result in more joined entries and thus smaller tables, at the price of diluted latency distance values. Whereas, a lower σ value should result in very few joins and thus more linear Compass Tables, with more accurate latency entries. Finding an similarity threshold value which guarantees optimal latency and sub-linear Compass Table sizes is not a trivial task. In general, the optimal σ value strongly depends on the execution environment characteristics. As we shall see in the evaluation chapter of this thesis the biggest data lookup performance can be achieved for environments which are evenly composed of mobile and static nodes. Given such configurations σ has to be additionally experimentally evaluated for various combinations so as to yield the same lookup performance for reduced Compass Table sizes.

5.1.3 Reliable Data Flows

As Figure 5.1 suggests failures of nodes is a common occurrence in distributed systems. While the Safety-Ring helps to overcome node failure issues of the control flow the

failures related to the data flow such as lost data items, lost service instance state etc. necessitate special handling.

So far the passive-standby recovery has been identified as the ideal approach to data flow reliability due to its conservative utilization of node resources. Traditional passive-standby approaches base their recovery strategies on limited redundancy, i.e., one backup node only. Simply put, the traditional passive-standby approach assumes the backup node to be ideal and not endangered by failure in any situation. However, in heterogeneous environments no node can be assumed as ideal, all nodes have a tendency to fail or at least become unresponsive to the others for a certain period of time. This is especially the case if they are mobile or resource limited nodes are involved in the system.

To increase the robustness of passive-standby in heterogeneous environments the most straightforward approach is to extend the redundancy of the checkpoints beyond the ideal backup node. By increasing the number of backup nodes for the state checkpoints of an active service instance, the robustness to node failures of the data flow will be increased. With more backup nodes at hand, continuous data flow can be rerouted to one of the backup nodes. On the other hand, increased backup redundancy entails the assertion of the state consistency among the backup nodes as well. As in the case of workflow instances, only if all backup sites feature the exact same state, correct recovery is possible in face of failures. Recovery of node failures based on inconsistent data can result in unforeseeable consequences.

Although it features a scalable, reliable and consistent data store at its core the Safety-Ring cannot be used for streaming instance state checkpoints for reasons of efficiency and compatibility as elaborated in the follow-up:

- The Safety-Ring applies the *symm* mapping function of symmetric replication (Section 3.2.3, Definition 3.10) so as to locate the replication SR-nodes. The *symm* function is agnostic to the availability of service instances at the replicated nodes. This implies that state checkpoints would predominantly end up at SR-nodes that do not feature the backed up service type in storing at the Safety-Ring. Hence, an additional data replication step would be necessary so as to replicate the backed-up state to the replacement service instance in the event of failure. The additional transfer of state from the Safety-Ring to the optimal backup node would delay the recovery time.

- The communication overhead, in terms of exchanged messages, of Paxos commit could turn out to be too much of an overhead for resource limited (e.g., networking capabilities) active service instances.

- The fault handling of streaming service instance failures would have to be delegated to the SR-nodes, which would correspond a considerable interference with the semantics of the passive-standby approach.

For all the mentioned reasons the Safety-Ring should be avoided for replication of streaming instance state. Rather, the state backups should be made available at all backup nodes directly. To ensure consistency of the check-pointed state at all distributed backup nodes a more lightweight consistency protocol should be considered such that

it can even be carried out by resource limited devices. Going back to the discussion of Section 5.1.2 where we assume memory and CPU resources to be negligible even for mobile devices we chose to apply a consistency protocol that conserves bandwidth resources instead. Being the most basic distributed transactional protocol, with the smallest communication overhead we opt for the two phase commit (2PC, Section 3.3.2) so as to enforce consistency at the distributed backup nodes.

Redundant Passive-Standby

The big idea behind the redundant passive-standby approach is to increase redundancy of an active service instance state checkpoint by simultaneous replication to multiple backup nodes. To ensure consistency of the replication process it is supported by means of a 2PC distributed transaction. In the context of 2PC, the active streaming instance becomes the transaction coordinator whereas the backup nodes become the transaction participants. Along the lines of the 2PC protocol, the active service instance interacts with the backup nodes in two phases:

- In the *Voting* phase the active streaming instance (i.e., coordinator) sends the its state checkpoint along with the prepare request to all backup nodes. The backup nodes take over the checkpoint state asses the prepare request of the coordinator by means of the local concurrency control mechanism and respond accordingly.
- In the *Commit* phase the coordinator issues the commit request if all backup nodes are ready. Once all backups have acknowledged the commit request the coordinator acknowledges the backup to its upstream service instance just as in the case of the traditional passive-standby approach.

For all the other cases the current transaction is aborted and has to be retried at a later stage in time by the coordinator. In the meantime, the upstream service instance has to wait for the acknowledgment from the downstream service instance. Figure 5.9 illustrates the 2PC enabled passive-standby checkpointing.

Since in our model there are no stable nodes backup nodes can fail themselves. That is why we enable the active streaming instance with capability of selecting the backup nodes. In case a transaction could not be competed due to backup node failure, a new backup node is selected inside the environment by the coordinator and the a new transaction is initiated. Naturally, the active service instance will select r backup nodes that have the same service type already deployed at them. Moreover, out of the set redundant and inactive service instances the active one will select the r best ones w.r.t. their metadata characteristics. Precisely, the r best service instances according to the routine `getSI(`a_{next}`)` (i.e., Algorithm 4.5) will be selected.

For all of its benefits (e.g., simplicity and efficiency) 2PC is commonly known to be vulnerable to the transaction coordinator failure. Recap, in case the coordinator of a transaction fails in the course of the *voting* phase after having issued the prepare requests, the transaction will block. This issue can however be effectively overcome by exploiting the failure recovery strategy of passive-standby. Since the active streaming instance is at the same time also the coordinator the upstream node will detect its absence (e.g., due to failure) and can thus terminate the ongoing transaction (if any) on

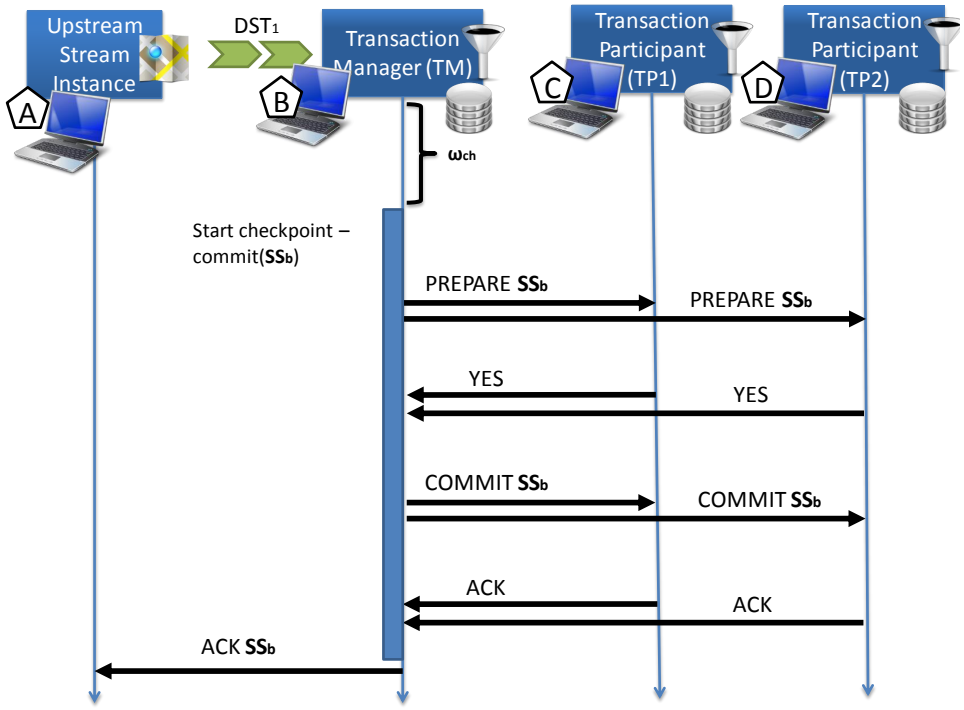


Figure 5.9: Passive-standby checkpointing in the context of 2PC transactions.

its behalf. In recovering the interrupted data flow at a backup node, the replacement service instance also becomes the new transaction coordination for future reference.

In order to be able to resolve any ongoing transaction of the downstream service instance, the upstream service instance has to be aware of the downstream transaction state, i.e., the current downstream coordinator, the current downstream checkpoint and the current downstream backup nodes. Only when this information is available at the upstream streaming instance the downstream coordinator can be successfully recovered. The downstream coordinator transaction state is obtained along with an acknowledgment message of the downstream node. Figure 5.10 depicts the recovery of a failed a transaction coordinator, i.e., active streaming instance.

Example 5.9

As we can observe from Figure 5.10 there are five peers (A, B, C, D, E) in the environment such that at nodes A and B active streaming instances are residing. Thereby, A is the upstream streaming instance of node B which is downstream streaming instance in streaming to it the data stream DST_1. The nodes C, D and E feature redundant service instances of in terms sink service type (depicted with the same icon) of node A. In doing so, the nodes C and D serve as backup nodes for node B.

As further illustrated in this figure node B fails after having received the commit readiness from C and D which blocks the current transaction denoted with the identifier CH_5. Once the upstream service instance at node A detects the absence of acknowledgment messages from A it triggers the passive-standby recovery in the traditional sense by rerouting the data stream DST_1 to node C which becomes the new down-

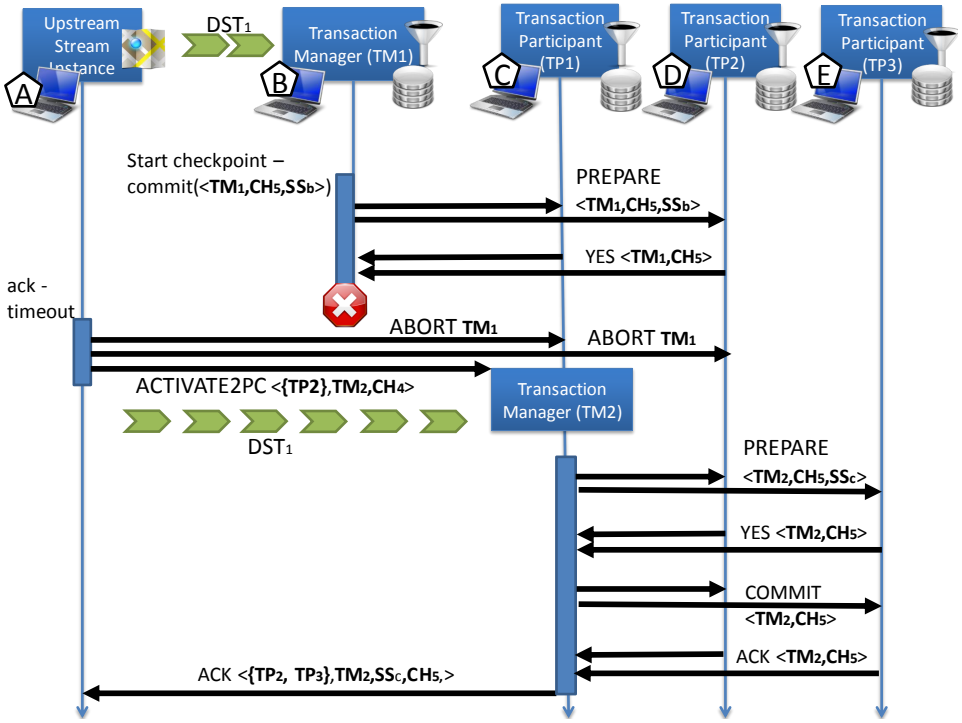


Figure 5.10: Redundant Passive-standby recovery of a failed transaction manger.

stream streaming instance. However, for redundant passive-standby A also needs to abort any ongoing transaction for which B has been the coordinator. That is why it sends an $ABORT$ request to all backup nodes. Moreover, node A also needs to activate the new coordinator, hence it sends to C the last checkpointed transaction state it is aware of, i.e., the remaining backup nodes, the new coordinator identifier and the last transaction identifier – $\langle \{TP_2\}, TM_2, CH_4 \rangle$. This information the upstream streaming instance obtained with the last acknowledgment message.

Once activated the new active streaming instance at node C will eventually start a checkpoint of its own with a new state SS_c and a new 2PC transaction identifier CH_5. But before it can do so, it has to pick a new backup node, which is node E, so as to reinstate the replication factor of $r = 2$. Upon successful completion of the transaction CH_5 node D will send an acknowledged message to the upstream service instance asserting the latest transaction checkpoint state along with the latest backup nodes, the latest coordinator and the latest transaction identifier, and the most recent state of the service instance checkpoint – $\langle \{TP_2, TP_3\}, TM_2, CH_5, SS_c \rangle$.

As the previous example already suggests to fault-handle downstream service instance failure, the upstream streaming instance needs to be always alive for the redundant passive-standby to work. This requirement is however inherently granted by the passive-standby idea itself. Precisely, the upstream service instance will also be backed by another upstream service instance and a set of redundant backup nodes. These will jointly fault-handle the upstream service instance if it should fail. To cope for upstream service instance failures, the last acknowledged downstream service instance

state has to be stored redundantly, i.e., shared with the backups, as well. Should the upstream service instance fail, its backup will contain the backed up acknowledgment of downstream service instance and can fault-handle from that state on downstream node failures. Any checkpoint of an active streaming instance should thus feature its current stream processing state and the latest acknowledgment of downstream service instance. Hence, we define a checkpoint of an active streaming instance for the redundant passive-standby approach to be as follows:

Definition 5.3 (Redundant passive-standby checkpoint – ch)**.** *A checkpoint ch of the redundant passive-standby approach for a streaming instance $si \in SI_c$ is defined as tuple $ch(si) = (ss, cnt_{ch}, cnt_{co}, ack_{si}, ack_{co}, ack_{ch}, SI_{ack})$ where:*

- *$ss \in SS$ is the current state of si with:*

$$ss = si.s.ss$$

- *$cnt_{ch} \in I\!N^+$ is the current checkpoint state counter of si,*
- *$cnt_{co} \in I\!N^+$ is the current coordinator counter of si,*
- *$ack_{si} \in SI$ is last acknowledged downstream coordinator of 2PC,*
- *$ack_{co} \in I\!N^+$ is the last acknowledged counter of the downstream coordinator,*
- *$ack_{ch} \in I\!N^+$ is the last acknowledged counter of the downstream checkpoint,*
- *$SI_{ack} \subset SI$ is the last acknowledged current downstream set of backup streaming instances.*

Whereas the set of all redundant passive-standby checkpoints for all active streaming instances is denoted as CH. □

Observe that ack_{si} and ack_{co} together denote the downstream transaction coordinator in the context of Example 5.9, whereas ack_{ch} denotes the transaction identifier.

The checkpoint *ch* is replicated to the backup nodes is supported with a 2PC distributed transaction. For this to work the checkpoint *ch* has to be embedded into a distributed transaction *item* and back. To this end the mapping functions f_{cd} and f_{dc} are introduced as follows:

$$f_{cd} : CH \mapsto DI \wedge f_{cd} : DI \mapsto CH$$

Algorithm 5.8 shows the 2PC embedded checkpointing approach into passive-standby for multiple backup nodes.

Note, that in case of a successful commit the routine `backup()` will return the acknowledgment as a complex data structure and not send the acknowledgment itself. The actual sending of the acknowledgment data structure to the upstream streaming instance takes place in Algorithm 4.1, line no.13 where the returned data structure of `backup()` is just appended to the actual acknowledgment message. Moreover, the timestamp of the transaction is determined by the current counter of the checkpoint. This can be attributed to the fact that the our 2PC distributed transactions apply an optimistic serializability approach to concurrency control. Even though 2PC is

Algorithm 5.8 $si.backup(ss_{new}, SI_{ch})$. Updates the checkpoint state of si to the given latest state ss_{new} and an increased checkpoint counter cnt_{ch}. Afterwards it creates a 2PC distributed transaction dtx with the checkpoint of si, i.e., $ch(si)$, to be the transaction item. Then, the transaction dtx is started such that all service instances of SI_{ds} are the transactional participants. In case dtx could be committed it updates the internal service instance state to ss_{new} and the new counter, upon which all upstream service instances are acknowledged with the current coordinator si, its counter cnt_{ds} and the backup nodes set SI_{ds}. For all other cases the backup is not performed no acknowledgments are issued.

1: $ch(si) \leftarrow CH(si)$
2: $ch(si).cnt_{ch} \leftarrow ch(si).cnt_{ch} + 1$
3: $ch(si).ss \leftarrow ss_{new}$
4: $item \leftarrow (f_{cd}(ch(si)), W)$
5: $dtx \leftarrow (''RANDOMKEY'', ch(si).cnt_{ch}, \{item, C\}, \{(item, C)\}, 2PC)$
6: $Participants \leftarrow SI_{ch}$
7: **if** $n.commit(dtx) = COMMIT$ **then**
8: $CH \cup ch(si)$
9: $retVal \leftarrow (si, ch(si).cnt_{ch}, ch(si).cnt_{co}, SI_{ch})$
10: **return** $retVal$
11: **end if**
12: **return null**

commonly used in conjunction to pessimistic locking (e.g., strict two phase locking – SS2PL [WV01]), redundant passive-standby follows another approach.

Since only the active service instance issues transactional writes, concurrent transactions at the backup nodes should not exist, thus the probability of transactional conflicts is very low. Preemptive locking of the backup nodes thus would unnecessarily consume additional network resources in acquiring the locks. To save network resources, the 2PC transaction protocol of redundant passive-standby assumes asynchronous replication of checkpoints to the backup nodes in conjunction optimistic serializability.

However, our reliability model does not completely rule out concurrency of transactions by default. Concurrent transaction can occur when the upstream streaming instance introduces the a new coordinator which consequently starts to checkpoint its state. In case the downstream streaming instance is still alive but has been recovered by the upstream service instance due to a false failure suspicion then concurrency will be introduced. That is, there will be two coordinators at the same time issuing transactional writes to the backup nodes. In face of concurrent writes our conflict resolution model always decides for the transaction with the latest transaction coordinator first and for the transaction with the latest transactional timestamp second. Algorithm 5.9 provides the conflict resolution of optimistic serializability for the 2PC transaction protocol of the redundant passive-standby approach.

Note, that Algorithm 5.9 overrides the routine `readyToCommit` already introduced in Algorithm 3.16, in the context of 2PC prepare acknowledgment, so as to enhance the concurrency control mechanism for optimistic serializability.

Algorithm 5.9 *tp.isCommitReady((dtx)* Given the transaction dtx extracts from it the checkpoint $ch(si)$ for the coordinator si. Then it checks if the coordinator counter of $ch(si)$ is greater than the current coordinator counter cnt_{ch} of the participant tp. If this is the case, then there is a new coordinator and the transaction is accepted. Otherwise, it checks if the checkpoint counter of $ch(si)$ is greater or equal than the checkpoint counter cnt_{ch} of tp. If this is the case, then there is a new checkpoint and the transaction is accepted. If both case do not apply an outdated transaction is trying to commit, thus it has to be aborted.

1: $\exists item \in dtx.Item : item \neq C$
2: $ch(si) \leftarrow f_{dc}(item.di)$
3: **if** $ch(si).cnt_{co} > cnt_{co}$ **then**
4: $\quad cnt_{co} \leftarrow ch(si).cnt_{co}$
5: $\quad cnt_{ch} \leftarrow ch(si).cnt_{ch}$
6: $\quad$ **return true**
7: **end if**
8: **if** $ch(si).cnt_{co} = cnt_{co} \wedge ch(si).cnt_{ch} \geq cnt_{ch}$ **then**
9: $\quad cnt_{ch} \leftarrow ch(si).cnt_{ch}$
10: $\quad$ **return true**
11: **end if**
12: **return false**

Whenever a failure is detected at the upstream node the data stream is redirected to the optimal backup node (`getSI()`) as in the case traditional passive-standby. Since the failed service instance could have been in the process of coordinating an ongoing transaction the replacement coordinator will introduce concurrency with its transactional writes. To put the new coordinator into a better position, from a conflict resolution point of view, the upstream streaming instance assigns the recovery coordinator with a greater coordinator counter cnt_{co}. Thereby, the upstream service instance will set the new coordinator value such that it correspond to an increased value of the last acknowledged coordinator counter (i.e., ack_{co}). Algorithm 5.10 shows the passive-standby embedded coordinator recovery.

We assume also that canceled transaction can also occur due to the failure of participants. In such situations the failed participant is replaced with a new one and a new transaction is issued in a later point in time. Since service instances that feature sources of the data stream do not have an upstream node they cannot be recovered by means of redundant passive-standby. Therefore, we have to assume the streaming instances featuring the data source to be ideal (i.e., fail-safe) as well. For any other service instance failure in the data flow, redundant passive-standby will perform a recovery – provided the number of simultaneous failures does not exceed the replication factor, i.e., the number of backups. We additionally assume that the streaming instance state of each checkpoint is not too big, in terms of data size, and fine grained such that replication of it across a set of redundant backup nodes would not be too resource consuming.

Given that no failures occur in the course of the coordination process, the approach of redundant passive-standby thus additionally burdens its traditional counterpart by just one phase of exchanged messages. That is, redundant passive-standby features the

Algorithm 5.10 $si.ackTimeout(si_f)$ In the event of service instance si_f failure redirects its output stream DST_s to the backup service instance si_{bk}. Selects si_{bk} to feature the highest checkpoint number cnt. Afterwards it activates enables si_{bk} for checkpointing with the backup node set SI_{ds} and the increased coordinator counter cnt_{ds}. Afterwards, it stops the failed service instance preemptively in case the suspicion was wrong. Finally, updates the corresponding workflow instance wi with si_{bk}.

1: $DST_s \leftarrow si.getStream(si.s.ss, si_f)$
2: $ch(si) \leftarrow CH(si)$
3: $si_{bk} \leftarrow si.getSI(ch(si).SI_{ack})$
4: $si.setStream(si.s.ss, si_{bk}, DST_s)$
5: $si_{bk}.activate2PC(ch(si).ack_{co} + 1, ch(si).ack_{ch}, ch(si).SI_{ack} \setminus si_{bk})$
6: $si_f.abort2PC(ch(si).ack_{co})$
7: **for all** $si_{bk} \in ch(si).SI_{ack} \setminus si_{bk}$ **do**
8: $\quad si_{bk}.abort2PC(ch(si).ack_{co})$
9: **end for**
10: $\exists wi \in WI : si \in wi.SI_a$
11: $wi.SI_a \setminus si_f$
12: $wi.SI_a \cup si_{bk}$

prepare and *commit* message exchange phase, in a failure-free scenario. In each phase r messages are sent back and forth between the participating nodes. Hence, only additional $4r$ messages are burdened on the system by our approach.

The checkpointing process itself is independent (i.e., asynchronous) of the data flow and does not affect of the throughput directly. It does however delay the subsequent checkpoint until the running one has been acknowledged by the distributed transaction. This implies that the delay in state checkpointing should result in more data items at the output streams of the upstream streaming instance that have not been acknowledged. Hence, our approach should affect the recovery performance of passive-standby. In the event of downstream coordinator failure, all unacknowledged data items at the upstream steaming instance will have to be resent to a backup so as to restore the lost state. Given more unacknowledged upstream data items the state recovery process at a backup node should thus last longer. However, for all of its benefits redundant passive-standby can be considered as efficient from a resource consumption (e.g., network bandwidth) point of view, such that it can be applied to heterogeneous environments.

5.2 Self-optimizing Workflow Definition Execution

A typical Peer-to-Peer execution environment is highly dynamic given the heterogeneous nature of the participating nodes. In view of available hardware resources, no two nodes tend to exhibit the same service instance execution behavior. While well-equipped nodes will feature stable and powerful performance, resource limited nodes will feature low performance and will be more likely to fail (e.g., due to resource depletion) than other nodes. Mobile nodes (if present), on the other hand, will likely change

their positions over time, and thus feature constantly changing data transmission characteristics. In view of service type deployment, some peers will host multiple service types while other peers will be dedicated to certain service types only.

Whatever the setting, all peers of a heterogeneous environment will feature at all times constantly changing execution performance characteristics. When distributed execution of workflow instances is subjected to such a heterogeneous node environment, the individual dynamic behavior of peers will have a significant effect on the overall system performance. In particular the throughput of workflow instances in the system will be affected by dynamics of peers. If in the course of workflow instance migration low performing nodes are selected over others then the throughput of the system is likely to be reduced and exposed to risk of node failure. For example, nodes that are frequently failing should be less preferred for late-binding decisions. Availability of service types additionally affects the throughput of workflow instances in the system. If highly demanded service types are instantiated too scarcely in the environment, workflow instances will have to queue for them and thus get delayed. This applies both for application as well as system services.

Therefore, the goal of our work is to increase the system performance, in terms of workflow instance throughput, by optimizing the control flow execution at runtime. Our approach is based on an improved late-binding of service instances that takes into account the dynamics of the environment to a greater extent. Precisely, we seek to achieve an effective load balancing among service instances by focusing on two main aspects of heterogeneous execution environments:

1. Well-performing nodes. Nodes that support a high throughput of workflow instances have to be preferred by the control flow at late-binding time. Such nodes typically feature a high execution performance due to abundant local (hardware) resources. Hence, they should be able to perform many concurrent invocations of the same service type and thus reduce the workload of the other nodes. To classify well-performing nodes various aspects of dynamic runtime behavior should be considered such as responsiveness, workload, available memory etc.

2. Service type availability. To support the throughput of workflow instances, an adequate amount of frequently used service instances has to be provided as well. In case the system does not offer enough instances of a service type, that is demanded by many workflow instances, it will likely from a bottleneck from a workflow instance migration point of view. To this end, new instances, of frequently used service types, should be dynamically created by the system. For the deployment of new instances well-performing nodes should be preferred so as not to overload the already resource limited ones. On the other hand, if the system features an abundance of instances of a service type that is not frequently used for late-binding, some of the instances should be decommissioned by the system. This way local resources will be conserved and if necessary more demanded service types can be deployed instead.

Traditional workflow engines rely on centralized metadata aggregation on the execution environment in conjunction to centralized decision making. The price of such an approach is limited scalability (as elaborated in Section 1.3, Example 1.4). Literature

(e.g., [STS+06]) provides confirmed reports of scalability bottlenecks when centralized late-binding is applied to scalable execution environments.

Contrarily to the traditional engines, our distributed execution model is completely decentralized and autonomously orchestrated by each orchestrator node itself. In our system each orchestrator is capable of autonomous late-binding of service instances, based only on locally available metadata. To support the orchestrator nodes in their decentralized late-binding functionality we choose to enrich them with additional information on the dynamic aspects of the execution environment. This in turn allows us to better classify the well-performing nodes.

Although additional metadata facilitates an improved detection of well-performing nodes, they might turn out not be that supporting for throughput as they lack the frequently used service types. Therefore, we enable the orchestrator nodes with hot-deployment functionalities for any kind of application service type. This in turn allows us to boost the throughput for sluggish workflow instances provided that service type scarcity is the main problem.

5.2.1 Extended Metadata on the Execution Environment

In order to identify the well-performing nodes at a certain point in time all aspects of their execution behavior have to be considered, especially the dynamic ones. The more information we have on nodes the better decisions we can make at runtime while entrusting them with important system functionalities, such as Safety-Ring participation, activity invocations etc.

Every node of the execution environment usually exhibits static and dynamic execution behavior at runtime. These characteristic can be summarized as follows:

- *Static* – Static behavior of nodes can be described with characteristics that either do not change at all or rarely change, over a longer period of time. For instance, hardware equipment can be considered as a static characteristic that can help classify nodes w.r.t. performance. However, such characteristics are generally concealed behind the service type interface and cannot easily accessed by the system.

- *Dynamic* – Dynamic behavior of nodes can be described with characteristics that do frequently change over a longer period of time. Unlike static characteristics, dynamic characteristics can be disclosed by means of external measuring (e.g., by other nodes) and thus statistically maintained by the system. For example, node positioning, workload, available resources, latency, stability etc. can to some extent at runtime be measured by other nodes and shared with the system so as to obtain a better insight on the execution environment. To grasp the dynamic behavior of nodes, all it takes is to perform periodic probing among themselves and to report the values to the system. Dynamic node characteristics that can not be externally measured, can nevertheless be retrieved by enforcing the system nodes to periodically update the workflow engine with their internal dynamic characteristics. For example, currently available resources and physical location can be submitted to the workflow engine.

Existing metadata, such as hosts workload, i.e., MD, cannot reliably tell us whether nodes are suitable for late-binding. For instance, the least loaded node could be a mobile one that is located far away and thus is susceptible to failure and low data transmission. To capture some extent of dynamic node behavior at runtime, we introduce a set of additional node metadata so as to power the system services. Precisely, we introduce next to the already existing workload metadata the *stability*, *mobility* and *latency* metadata that are extended to the global metadata repository ρ. The additional metadata sets provide us with further insight on nodes execution aspects such as their failure frequency, and responsiveness, which certainly tend to frequently change over time.

Stability Metadata

To capture the dynamic behavior of nodes w.r.t. failure frequency the stability metadata is introduced. By statistically maintaining the number of reported failures per node over a certain period in time, the workflow engine can classify each node, in terms of its stability. Hence, instable nodes are concluded to be more likely to fail again and can thus be avoided (if possible) over other more stable nodes at runtime. The stability of nodes is classified into six categories, i.e., *failure-free, very-stable, stable, instable, critical* and *failed* which have been experimentally classified as good descriptors of node runtime stability state. The node stability categories are represented by the category set CT_{st} as follows:

$$CT_{st} = \{FAILUREFREE, VERYSTABLE, STABLE, INSTABLE, CRITICAL, FAILED\}$$

Each stability category is represented by a failure frequency threshold value, i.e., $tsh_{ff}, tsh_{vs}, tsh_{s}, tsh_{is}, tsh_{cr}$ and tsh_{fl}, which needs to be matched by a node so as to fall into the respective category. The exact threshold values of the stability categories are subject to end-user application specification. The threshold values are represented by the category threshold set TSH_{st} as follows:

$$TSH_{st} = \{tsh_{ff}, tsh_{vs}, tsh_{s}, tsh_{is}, tsh_{cr}, tsh_{fl}\}$$

The formal definition of the stability repository is provided as follows:

Definition 5.4 (Stability metadata– st)**.** *The stability metadata for node a $n \in N$ at timestamp $t \in T$ is defined as tuple $st(n,t) = (f_{st}, \tau_{st}, \omega_{st}, ct_n)$ where:*

- $f_{st} \in I\!N^{+}$ *is current failure counter,*
- $\tau_{st} \in T$ *is the timestamp of the first failure report,*
- $\omega_{st} \in T$ *is the failure frequency evaluation window size,*
- $ct_n \in CT_{st}$ *is current stability category value,*

The set of all stability metadata for node n is defined as $ST(n)$, whereas the set of all mobility metadata for all nodes is defined as ST. □

The stability metadata set is an extension to the global metadata repository ρ. Hence, all peers should also be subscribed to the stability metadata at all times, as well.

$$\rho \cup ST$$

However, only the SR-nodes get to update the stability repository as they are the ones to detect the failures of nodes. Whenever a SR-node suspects a node to have failed it sends an update to repository featuring the suspected node and the failure timestamp (i.e., Algorithm 5.3, line no.3). In turn, the repository acknowledges the failure report and updates the frequency metadata. Based on end user specified failure frequency thresholds (i.e., TSH_{st}) and evaluation windows (i.e., ω_{st}) the reported nodes are categorized into the corresponding stability category. Algorithm 3.26 shows the node failure report handling.

Algorithm 5.11 *n.updateStatistics(n', τ)* Creates a new stability report $st(n', \tau)$ for given node n' based on the latest report $st(n', \tau_{max})$. It further increases the frequency counter. If the given failure timestamp τ exceeds the evaluation window ω_{st} as compared to the timestamp of the first occurrence τ_{st} it categorizes n' w.r.t. thresholds values of TSH_{st}.

1: $\exists st(n', \tau_{max}) \in ST(n') : \nexists st(n', t_x) \in ST(n')$ **where** $t_x > \tau_{max}$
2: $st(n', \tau) \leftarrow st(n', \tau_{max})$
3: $st(n', \tau).f_{st} \leftarrow st(n', \tau).f_{st} + 1$
4: **if** $\tau - st(n', \tau).\tau_{st} \geq st(n', \tau).\omega_{st}$ **then**
5: $st(n', \tau).\tau_{st} \leftarrow \tau$
6: **for** $tsh \in TSH_{st}$ **do**
7: **if** $st(n', \tau).f_{st} < tsh$ **then**
8: $st(n', \tau).ct_n \leftarrow n.category(tsh)$
9: **return**
10: **end if**
11: **end for**
12: **end if**
13: $ST(n') \cup st(n', \tau)$

Note, that the node failure reports bear a certain uncertainty w.r.t. the provided failure timestamp τ that is reflected in the unsynchronized time clocks of the individual nodes. That is, each node reports a failure timestamp τ based on its own internal clock. In practice, the internal clocks of any two nodes can be considered as perfectly synchronized – the same. This is however, negligible in the case of the stability repository as we are not interested in the exact time of failure for each node. Rather, we are interested in the stability category value for each node which is based on an approximation of the node failure frequency. The node category value provides the orchestrator nodes with sufficient information so as to identify stable nodes.

Mobility Metadata

To capture the dynamic behavior of nodes w.r.t. movement the mobility metadata is introduced. By keeping track of the actual physical location per node (e.g., by means

of GPS device at each node), relative physical distances among nodes can be computed and thus movement deduced. Based on the computed relative physical distances among nodes, service instances can be identified that feature nodes which are closer by than others. Physically closer nodes, in case of inherent mobility, should be less likely to move out of the reception range from each other. As a consequence they will be less likely mistaken to have failed. Frequently moving nodes, on the other hand, are more likely to get damaged and thus fail. This implies that physically closer nodes should be favored over distant ones at late-binding time. By classifying the nodes into mobility categories, such as in the example of the stability repository, relative physical distances cannot be captured. To specify the physical location of nodes, we leverage the three dimensional Cartesian coordinates system. The formal definition of the mobility metadata is provided as follows:

Definition 5.5 (Mobility metadata – m)**.** *The mobility metadata of a node $n \in N$ at timestamp $t \in T$ is defined as tuple $m(n,t) = (x,y,z)$ where $x,y,z \in \mathbb{R}$ are the Cartesian coordinates. The set of all mobility metadata for node n is defined as $M(n)$, whereas the set of all mobility metadata for all nodes is defined as M.* □

Given that the physical coordinates of a node are lying in the Euclidean space, the distances among nodes can be computed by means of the Euclidean distance function f_m as follows:

$$f_m : M \times M \mapsto \mathbb{R}^+ \textbf{ where}$$

$$f_m(m_a, m_b) = \sqrt{(m_a.x - m_b.x)^2 + (m_a.y - m_b.y)^2 + (m_a.z - m_b.z)^2}$$

The mobility metadata set is an extension to the global metadata repository ρ. Hence, all peers should also be subscribed to the metadata metadata at all times, as well.

$$\rho \cup M$$

However, since node physical location cannot be easily externally assessed by other nodes, each node itself has to publish its physical coordinates to the system. This is done by each node on a periodic basis to the global metadata repository ρ.

Latency Metadata

Latency metadata is introduced so as to capture the dynamic behavior of nodes w.r.t. to their capability to transmit data and to some extent performance. By keeping track of the relative round trip delay times among nodes, data lookup times can be derived. In the context of service instance late-binding, lower latency nodes should be preferred when the workflow instances are to be migrated. By late-binding low latency nodes only at each migration step the workflow instances should complete faster and thus the overall throughput performance of the workflow engine improved. Performance of nodes to can be some extent deduced from latency information as well. That is, more powerful and underutilized nodes should immediately forward workflow instances (also data) that is traversing them, whereas congested (i.e., overloaded) and resource limited nodes should take more time.

To have a clear notion about latency, each node has to measure other nodes of interest by means of periodical probing. Given a huge and scalable node environment latency probing of all nodes is uneconomical from a resource consumption point of view. If the majority of a scalable execution environment is to be probed each node has to raise significant storage and network bandwidth resources to perform probing. Mobile nodes are usually not up to such a challenge as they can offer only limited resources. Hence, extensive measuring of huge sets of nodes is not an option for heterogeneous execution environments. The set of nodes to be probed by each node has to be scaled down, without completely loosing the knowledge on latency values to the nodes that are not directly probed.

Since orchestrator nodes are only interested in latency so as to identify nodes which feature better responsiveness over others we can apply node distance descriptions instead. A notable approach to distance based nodes description is the one of Lipschitz [JL84]. In this approach a significantly smaller subset of all nodes, referred to as the *landmarks* set, is used to compute latency distances against. The set of landmark nodes is defined as LMK as:

$$LMK \subseteq N$$

Each node probes its latency towards the landmark nodes and declares the returned values as a n-dimensional coordinate of the Euclidean space. Thereby, each dimension corresponds the latency distance value towards one of the landmark nodes. To measure the latency value towards a landmark the function f_{msr} is introduced as follows:

$$f_{msr} : N \times LMK \mapsto I\!R^{+} \cup \{0\}$$

The landmark distance coordinates of a node $n \in N$ are defined with the vector **lv[n]** as follows:

$$\textbf{lv[n]} = [f_{msr}(n, lmk_1), ..., f_{msr}(n, lmk_i), ...)] \textbf{ with } lmk_i \in LMK$$

The set of all landmark distance coordinates for all nodes is defined as LV:

$$LV = \bigcup_{\forall n \in N} \textbf{l[n]}$$

By expressing the nodes by means of landmark distance coordinates the global repository can be exploited to efficiently disseminate the coordinates as metadata to all nodes of interest. The metadata describing the latency coordinates in the context of landmarks distances is formally defined as follows:

Definition 5.6 (Latency metadata – lt)**.** *The latency metadata of a node $n \in N$ at timestamp $t \in T$ is defined as $lt(n,t) =$ **lv[n]** where **lv[n]** $\in LV$ is its landmark distance vector. The set of all latency metadata for node n is defined as $LT(n)$, whereas the set of latency metadata for all nodes is defined as LT.* □

Naturally, all peers are subscribed to the global metadata repository at all times for node latency coordinates. All peers also periodically publish their landmark coordinates to ρ as well.

$$\rho \cup LT$$

By exploiting the triangular inequality condition of the Euclidean space among any two nodes featuring distance coordinates towards the same landmarks pivots, their corresponding distance can be computed by means of the Euclidean distance function f_{lt} as follows:

$$f_{lt} : LV \times LV \mapsto I\!R^{+} \textbf{ where}$$

$$f_{lt}(\mathbf{lv}[n_a], \mathbf{lv}[n_b]) = \|\mathbf{lv}[n_a] - \mathbf{lv}[n_b]\|_2$$

Hence, given a set of node latency coordinates each node can compute by means of f_{lt} another node with the least distance towards it and thus exploit it for some workflow engine functionality, e.g., to invoke service instances.

Distance approximations of the Lipschitz approach improve with the size of the landmarks set. Hence, this approach is subject to a scalability-precision trade-off. In practice, devising landmark sets which are small and precise for distance approximations, at the same time is not trivial, which is the main drawback of the Lipschitz approach. Finding the right balance between scalability and precision is the topic of many works in literature, e.g., [TC03]. As for our work, we use landmark node sets of different sizes that are experimentally determined and are dependent on the size of the execution environment. The landmark node sets however in our case always contain the super peer as well as other stable nodes that do not feature any mobility, i.e., static nodes.

Combined Node Metadata

Given the additional metadata sets ST, M, LT the global repository ρ features an abundance of node information that needs to be meaningfully put into use in supporting the system services. In general, the orchestration service is always interested in the optimal successor service instance at workflow instance migration time. To find an optimal (i.e., well-performing) node out of the set of locally available nodes we need to consider all of its dynamic characteristics at run time. For instance, a node might be underloaded but instable at the same time, hence it should be avoided for workflow instance migration. Limited perspective on the execution environment affects the throughput of workflow instances. In order to be able to assess a node based on all of its dynamically changing characteristics (i.e., workload, stability, mobility and latency) subsets of the global repository ρ need to be meaningfully combined at the orchestrator nodes. Naturally, the subsets of ρ to be locally replicated are determined by the local subscriptions to succeeding activities.

To this end the local orchestrator node metadata set MD needs to be extended for the newly introduced dynamic node characteristics. The formal definition of the extended local metadata of an orchestrator node is defined as follows:

Definition 5.7 (Extended orchestrator service metadata – md_e)**.** *The extended execution environment metadata md_e locally maintained by the orchestrator si_o is defined by the tuple $md_e(si_o) = (md(si_o), \rho_{st}, \rho_m, \rho_{lt})$ such that:*

- *$md(si_o) \in MD(si_o)$ is local metadata according to Definition 4.13,*

- *ρ_{st} is the locally maintained subset of the stability metadata for the local subscription $md(si_o).sub_l(si_o)$, with:*

$$\rho_{st} = \{st(n,t) \in ST | \forall a_s \in md(si_o).sub_l(si_o).A_{succ} \; \exists (a_s, n) \in SI\}$$

- *ρ_m is the locally maintained subset of the mobility metadata for the local subscription $md(si_o).sub_l(si_o)$, with:*

$$\rho_m = \{m(n,t) \in M | \forall a_s \in md(si_o).sub_l(si_o).A_{succ} \; \exists (a_s, n) \in SI\}$$

- *ρ_{lt} is the locally maintained subset of the latency metadata for the local subscription $md(si_o).sub_l(si_o)$, with:*

$$\rho_{lt} = \{lt(n,t) \in LT | \forall a_s \in md(si_o).sub_l(si_o).A_{succ} \; \exists (a_s, n) \in SI\}$$

The set of all extended environment metadata for one orchestrator service instance is defined as $MD_e(si_o)$, whereas the set of all metadata for all orchestrator service instances is defined as MD_e. □

Based on the locally available extended metadata set $MD(si_o)$, an optimal node has to be found out of the set of all available nodes. For instance, at runtime the orchestrator has to find the optimal service instance when queried with subsequent activities as in the case of a workflow instance migration (e.g., Algorithm 5.1, line no.10). Thereby, each node is represented with its current value in terms of workload, stability, mobility and latency. All of these features of node dynamics have to be considered while selecting the optimal node. The solution behind our approach lies in node ranking. That is, for each node a rank is assigned based on its current metadata dimension values. Once all nodes have been ranked the one with the highest rank is picked so as to be the optimal node at the given point in time. Likewise, the node with the lowest rank can be chosen as well in case bad performing nodes are preferred.

The node ranking procedure is conducted in three steps as follows:

1. Candidate nodes extraction – For a given a subsequent activity a_n at the orchestrator si_o all candidate nodes that feature an instance of a_n have to be filtered out of the local metadata set $MD(si_o)$. The node candidate set is denoted as $Cnd(a_n)$.

2. Metadata extraction. Local metadata has to be extracted out of the local metadata set $MD(si_o)$ that describes the candidates of Cnd w.r.t. load, stability, mobility and latency. The candidate metadata is denoted as L_a for the load, ST_a for stability, M_a for mobility and LT_a for latency.

3. Feature vector assignment – Given the candidate set Cnd and the sets L_a, ST_a, M_a and LT_a, feature vectors have to be assigned to each node. Each dimension of the feature vector will reflect a node value of the metadata sets L_a, ST_a, M_a, LT_a for the given timestamp t.

In order to be able to extract the candidate nodes in first we have to find a local metadata element that features a subscription to given activity a_n. Thereby, multiple metadata elements might likely exist that feature subscriptions to a_n. This can be attributed to the fact that for each workflow definition a subscription is created, thus if different workflow definitions feature the same activity one metadata elements will be created per workflow definition at a control flow preceding node. Due to pub/sub replication by the repository ρ we assume that the redundant metadata elements will consistently feature the same candidate service instances of type a_n and the same metadata for them. Therefore, to the retrieve all node candidate is sufficient to first retrieve just one metadata element $md_e(si_o)$ of subscription to a_n. Expression 5.3 shows the extraction of the metadata element $md_e(si_o)$:

$$\exists md_e(si_o) \in MD_e(si_o) : a_n \in md_e(si_o).md(si_o).sub_l.A_{next}. \tag{5.3}$$

Now, that we have the metadata element we can extract the candidate nodes set from all service instances as shown by the Expression 5.4.

$$Cnd(a_n) = \{n \in N | \forall n \in N \exists (a_n, n) \in SI\} \tag{5.4}$$

In case $md_e(si_o)$ features only a subscription to activity a_n then $md_e(si_o)$ contains already all metadata and all candidate nodes we need to evaluate by default:

$$|md_e(si_o).md(si_o).sub_l.A_{next}| = 1$$

However, if the preceding activity of the control flow was a forking activity then the contained metadata of $md_e(si_o)$ might feature metadata on service instances that are not of type a_n. To this end we have to extract all metadata out of $md_e(si_o)$ such that it reflects the candidate nodes of $Cnd(a_n)$ only. Since local metadata is partitioned into the subsets $\rho_l, \rho_{st}, \rho_m$ and ρ_{lt} of $md_e(si_o)$ we have to filter them out individually in the second step of the node ranking process. Metadata node extraction based on the candidate set is shown in Expressions 5.5-5.8.

$$L_a = \{h(n,t) \in md_e(si_o).md(si_o).\rho_h | \forall n \in Cnd(a_n)\} \tag{5.5}$$

$$ST_a = \{st(n,t) \in md_e(si_o).\rho_{st} | \forall n \in Cnd(a_n)\} \tag{5.6}$$

$$M_a = \{m(n,t) \in md_e(si_o).\rho_m | \forall n \in Cnd(a_n)\} \tag{5.7}$$

$$LT_a = \{lt(n,t) \in md_e(si_o).\rho_{lt} | \forall n \in Cnd(a_n)\} \tag{5.8}$$

Observe that we filter all metadata in the in Expressions 5.5-5.8 based on the same given timestamp t. Timestamp t denotes the point in time at which the late-binding decision is to be made by si_o, thus metadata of this timestamp can be considered only.

Now that we have the extracted metadata in the third step we need to assign it to each node. To this end we use vector notation such that each node is represented with a feature vector. Thereby, the feature vector will be characterized by four dimensions such that each dimension reflects a node's metadata value for each of the sets L_a, ST_a, M_a and LT_a. Each dimension is normalized so as to bound the values within the limits of $0 - 1$. Normalization is achieved by dividing metadata value of the node by the

maximum value of the corresponding set. If metadata values are members of sets that are unbounded (i.e., physical distance, latency distance) the maximal values have to be provided artificially. To do so an expert end-user that administers the workflow engine has to manually set the maximal normalization values so as to meet the classification needs of the high-level application.

The load dimension of a node's feature vector is obtained by representing its load value of L_a divided by the maximal value 100. The stability dimension of a node's feature vector is obtained by representing its stability category value of ST_a divided by the maximum value FL. The maximum value FL corresponds to the *FAILED* stability category value of CT_{st}. The mobility metadata featured by the set M_a is multidimensional, thus it has to be reduced to just one dimension if it is to be used for the mobility dimension of the feature vector. To this end we apply the distance function f_m so as to characterize each candidate node relative physical distance to the orchestrator node. This is more applicable from an orchestrator point of view, as gives insight on the distance towards the candidate nodes, so that closer nodes can be preferred at late-binding time. The obtained mobility distance value has to be normalized by the artificial maximum value $D_m \in I\!R^+$ which represents the maximal distance value we are considering. The same approach is used for the latency dimension of the feature vector as well. Since it is also multidimensional it is reduced to just one dimension by means of the distance function f_{lt} in relation to the orchestrator node. Simply put, each candidate node is characterized by its latency distance towards the orchestrator node for the latency dimension of its feature vector. The obtained latency distance value has to normalized by the artificial maximum value $D_l \in I\!R^+$ which represents the maximal latency distance value we are considering.

Now that we dimension values are normalized within the bounds of $0-1$ they are subtracted from the maximal value 1 due to the preference of nodes that are characterized with lower (i.e., lower workload, lower stability category, lower distance) values. By subtracting each dimension from 1 the corresponding nodes will be awarded with higher feature vector values and thus higher ranks. Expression 5.9 shows the assignment of the feature vector for a node n:

$$n_o = si_o.n$$

$$\mathbf{v}[n] = [\underbrace{1 - \frac{h(n,t)}{100}}_{Load}, \underbrace{1 - \frac{st(n,t).ct_n}{FL}}_{Stability}, \underbrace{1 - \frac{f_m(m(n_o,t), m(n,t))}{D_m}}_{Mobility}, \underbrace{1 - \frac{f_{lt}(lt(n_o,t), lt(n,t))}{D_l}}_{Latency}] \tag{5.9}$$

By applying the feature vector assignment for each candidate node we obtain the feature matrix $\mathbf{v}$ as summarized in Expression 5.10:

$$\mathbf{v} = [\ \mathbf{v}[n_1]^{\mathbf{T}}\ \mathbf{v}[n_2]^{\mathbf{T}}\ ...\ \mathbf{v}[n_i]^{\mathbf{T}}\ ...\ ]\ \textbf{with}\ n_i \in Cnd(a_n). \tag{5.10}$$

In order to adjust each feature vector dimension for importance in relation to the others we devise a normalized weighting vector $\mathbf{w}$. The individual feature vector weights are end-user specific and thus subject to change, however they have to sum up to the value 1. Expression 5.11 summarizes the weighting vector representation.

$$\mathbf{w} = [\omega_l\ \omega_{st}\ \omega_m\ \omega_{lt}]\ \textbf{where}\ \omega_l + \omega_{st} + \omega_m + \omega_{lt} = 1 \tag{5.11}$$

Finally, we obtain the node ranks as a vector **rv** out of the feature matrix and the weight vector by applying the dot product on them. Expression 5.12 summarizes the rank vector computation.

$$\mathbf{rv} = \mathbf{v^T} \cdot \mathbf{w^T} \tag{5.12}$$

The computed rank vector **rv** can be queried by the orchestrator, in the context of the `getSI(a)` routine, for the maximal value and thus the corresponding node so as to obtain an optimal service instance. Depending on the use case of the rank vector, minimal values can be also queried so as to obtain low performing nodes.

5.2.2 Dynamic Service Deployments

The distributed execution model can only work in the face of node failures as long as there are enough service instances in the environment to substitute the failed ones. This applies for both system service instances as well as for application service instances. Thus the intuitive decision would be to provide always the biggest possible amount of service instances for all existing service types. The alternative of too few service instances should otherwise result in lower throughput performance. For example, in the case of the Safety-Ring, the more SR-nodes there are, the better the robustness of the system and the better the workflow instance throughput should be in distributing the transaction processing overhead among them. Better throughput should also be achieved by providing more application service instances as well. The more application service instances we have in the system, the better load balancing decisions we can make at late-binding time.

Although a high number of service instances (of any type) should improve the throughput of the system it should also incur more overhead on the nodes. Each service type entails a certain CPU and memory footprint that has to be matched by any node that is to host it. For example, the Safety-Ring service implies additional overhead on the nodes with its ring maintenance and distributed transaction algorithms. However, the hardware requirements of service types can turn out to be too much of a strain for some type nodes. Especially nodes that feature limited resources and inherent mobility are susceptible for overload due to unmatched service type (e.g., hardware) resource requirements. This implies that all peers should be carefully assessed for the deployment of service types and not all of them are necessarily suitable for a wide spectrum of service type deployments.

Even if optimal nodes are selected (e.g., hand picked by an expert) for deployment initially, given the high dynamics of underlying execution environment (e.g., Peer-To-Peer, Cloud) the workflow engine cannot rely on static configurations of service type so as to guarantee performance at all times. Static configurations of service types at nodes have the potential to substantially affect the throughput performance as described by the following four cases follows:

- Over-provisioning of all service types – In case all existing service types are excessively configured at all nodes, the system should benefit from a high throughput performance for all kind of workflow definitions. The price of such an approach is significant resource consumption at all nodes independently of the sys-

tem load. For low numbers of concurrent workflow instances, excessive deployments should turn out to be highly uneconomical. Such configurations are only beneficial for systems that are loaded at all times with plenty of workflow instances and that feature powerful computational nodes.

- Over-provisioning of selected service types – In case specific service types are excessive deployed, the system should benefit from a high throughput performance for and a reduced resource consumption such that less equipped (hardware) nodes can be used. However, if new workflow definitions frequently enter the system the previous configurations of certain service types might quickly become obsolete and resource wasting. Such configurations are only beneficial if the system is dedicated to a limited number of workflow definitions only which are instantiated in high numbers.

- Under-provisioning of selected service types – In case specific service types are scarcely deployed, the system should benefit from a reduced resource consumption and high throughput performance if various kinds of workflow definitions exist. However, this is only possible if the scarcely deployed service types are used for late-binding by a small number of workflow instances. Such configurations are only beneficial if the system features a high number of workflow definitions that are only occasionally instantiated.

- Under-provisioning of all service types – In case all existing service types are scarcely configured at all nodes, the system should benefit overall low resource consumption at all nodes at the price of overall low system throughput performance. Such configurations are only beneficial if the system is composed of resource limited nodes only that do not run a big number of workflow instances independent of the their type.

Therefore, we conclude the throughput performance of the distributed execution model to be strongly dependent on the number of service instances in the system and the number of concurrent workflow instances that invoke them. To improve the throughput performance of our workflow engine we will enable it with the functionality of dynamic reconfiguration of the execution environment. This means that our workflow engine is capable of instantiating new and to deactivating existing service instances in the environment. In dynamically reconfiguring the execution environment the engine is guided by the goal of supporting the workflow instance throughput and saving node resources at runtime.

To perform dynamic reconfiguration of the environment the engine first has to find the optimal reconfiguration node, and then based on the current state of the environment either instate a new or deactivate an existing service instance. We separate the dynamic reconfiguration functionality however into two independent aspects. The one addresses dynamic reconfiguration of system service types while the other addresses dynamic reconfiguration of application service types.

In the context of system service types, we limit ourselves to the Safety-Ring service only. Being one of the most resource intensive (e.g., memory and bandwidth) services in the system the workflow engine will dynamically change number of SR-nodes in

the environment. This way the workload of processing the workflow instances in a transactional fashion will be evenly distributed among a dynamically changing set of SR-nodes. In the context of application service types, we enable the workflow engine to dynamically add/remove service instances of any type. This way the throughput of workflow instances will be evenly distributed among a dynamically changing set of application service instances (for any type) at late-binding time.

Since we are bound to the requirements of an ideal workflow engine (i.e., Section 2.4), dynamic reconfiguration of the environment has to be performed in a decentralized fashion so as not to affect scalability of the engine. The remainder of this section discusses how dynamic reconfiguration of the execution environment is achieved in a distributed fashion.

Elastic Safety-Ring

In order not to affect the distributed execution of workflow instances by static configuring of the Safety-Ring, it should be able to dynamically reconfigure its ring topology. Depending on the overall workload in the system, induced by the amount of concurrently running workflow instances, the Safety-Ring configuration should feature an adequate amount of SR-nodes so as to accommodate the workload evenly. Although the workflow instances will be evenly distributed among the SR-nodes inherently, the load per SR-node should anyway increase if the amount of workflow instances is increased as well. To cope with high lode new SR-nodes should dynamically join the Safety-Ring configuration in case a high number of workflow instances is run against the system. At times of moderate workflow instances load in the system, some SR-nodes should become unnecessary and thus leave the Safety-Ring. Simply put, the Safety-Ring should elastically adapt itself, by adjusting the participation of SR-nodes, to the current number of workflow instances.

To address the goal of Safety-Ring elasticity in a distributed fashion we rely on the self-organization concepts of the underlying ring topology. Precisely, we exploit the ring topology node joining and leaving routines that are powered by a basic autonomic controller. We employ a control loop that monitors the load of a Safety-Ring enabled node and in case of overload/underload applies a reconfiguration operation. Such reconfiguration operations include the addition of new SR-nodes – in case of permanent overload – and the removal of existing SR-nodes – in case of permanent underload. The Safety-Ring is thus at runtime elastically increased and decreased by the controller depending on the SR-node load.

The Safety-Ring controller is however decentralized. Each SR-node is enabled with a controller of its own that allows it to autonomously perform reconfiguration decisions. Much like the orchestrator nodes are in their orchestration functionality. All controllers at the SR-nodes are however consistent in the business logic they conduct. Enabling the SR-nodes with reconfiguration capabilities entails a number of difficulties:

1. Autonomous reconfiguration operations should not result in an over-provisioned system of SR-nodes so as to conserve resources.

2. On the other hand, autonomous reconfiguration operations should not result in an under-provisioned system of SR-nodes so as to obstruct workflow instance throughput.

3. Given the underlying ring topology, frequent joins/leaves of SR-nodes should be avoided so as to prevent churning of the topology.

While the first two issues can be effectively resolved by configuring the controllers at each SR-node consistently, the third issue necessitates special handling. In general a controller relies on a set of predefined threshold values that will help it to keep its configuration functionality within controlled bounds. Each Safety-Ring controller keeps track of the overload/underload frequency at each SR-node by means of counting and triggers its configuration operations given specific threshold values for minimum/maximum frequency and specific time windows for underload/overload occurrences. The controller configuration thresholds are summarized as follows:

1. $tsh_{Lmax} \in I\!N^{+}$ is the overload threshold value,

2. $\omega_{er} \in T$ is the threshold value for the overload evaluation window.

3. $tsh_{er} \in I\!N^{+}$ is threshold value of the overload frequency for which a ring expansion operation should be taken.

4. $tsh_{Lmin} \in I\!N^{+}$ is the underload threshold value,

5. $\omega_{cr} \in T$ is the threshold value for the overload evaluation window.

6. $tsh_{cr} \in I\!N^{+}$ is threshold value of the underload frequency for which a ring contraction operation should be taken.

By assessing the overload/underload situations for frequency and time periods (i.e., ω_{cr} and ω_{er}) oscillations can be dampened out and thus the first two issues of distributed self-configurations handled. Thereby, the dampening of the controllers strongly depend on the setting of the controller threshold values. As with all controllers, the thresholds need to be manually specified by a expert end-user administering (i.e., administrator) the workflow engine depending the self-configuration needs of the high-level application. Setting the controller threshold values by means of tools that are based on machine learning techniques is also an option.

To compare the threshold values of the controller each SR-node locally maintains statistics that reflect its overload/underload occurrences. The SR-node load statistics are formally defined as follows:

Definition 5.8 (SR-node load statistic – $stat_{sr}$)**.** *Safety-Ring load statistics that are locally maintained by the SR-node si_{sr} are defined as tuple $stat_{sr}(si_{sr}) = (f_{er}, \tau_{er}, f_{cr}, \tau_{cr})$ such that:*

- $f_{er} \in I\!N^{+}$ *is current overload counter,*
- $\tau_{er} \in T$ *is the timestamp of the first overload occurrence,*
- $f_{cr} \in I\!N^{+}$ *is current underload counter,*

- $\tau_{cr} \in T$ *is the timestamp of the first underload occurrence,*

The set of all load statistics for all SR-nodes is defined as $STAT_{sr}$. □

The business logic of the controllers is simple and consistently applied across all SR-nodes. Precisely, the controller counts the number of times the specified threshold value (i.e., tsh_{Lmax} or tsh_{Lmin}) has been exceeded. If the frequency of excesses is higher than the specified configuration threshold value (i.e., tsh_{er} or tsh_{cr}) for the specified time window (i.e., ω_{er} or ω_{cr}) a reconfiguration situation is ascertained. Algorithm 5.12 shows the business logic of the controller.

Algorithm 5.12 *n.checkFrequency*(f, τ, ω, tsh) Increases the given frequency counter f by one. Updates the given timestamp τ to current time if it's the first count. If the difference between the current timestamp and the first occurrence timestamp exceeds the given window ω, the frequency counter f is evaluated. A higher value of f as compared to the given threshold *tsh* results in a reconfiguration situation and thus a *true* outcome. Otherwise $false$ is returned.

```
1: f ← f + 1
2: if f = 1 then
3:     τ ← n.currentTime()
4: end if
5: Δτ ← n.currentTime() − τ
6: if Δτ ≥ ω then
7:     if (Δτ/ω) f ≥ tsh then
8:         return true
9:     else
10:        f ← 0
11:    end if
12: end if
13: return false
```

The controller aids the SR-nodes in informing them when to reconfigure the topology, but it does not tell them how reconfigure the system. Reconfiguration of the Safety-ring is dependent on the underlying Chord ring topology and needs special handling. For example, frequent and concurrent SR-node join/leave operations have the potential to partition the ring into multiple disjunct ones [KEAAH05, RGRK04] which cannot be reconciled without manual intervention.

The approach of Safety-Ring to decentralized reconfiguration based on autonomous reconfiguration operations at each of the SR-nodes that are specific to the underlying ring topology. Having only a limited overview of the whole topology – the predecessor, the successor and the finger nodes – each SR-node should only change part of the ring that it is aware of. Its reconfiguration operations should not affect distant parts of the ring topology and SR-nodes that are associated it. To prevent the reconfiguration operations that interfere with nonadjacent SR-nodes we restrict each SR-node to changing parts of the ring that address its area of responsibility only, i.e., its key identifier space. In the context of ring expansion, this implies that a SR-node can add a new SR-node

only if the new SR-node would be located inside its key identifier space, i.e., if it would its predecessor. In the context of ring contraction, this implies that a SR-node can not remove any SR-nodes but only voluntarily leave the ring.

In face of concurrent reconfiguration operations by all the SR-nodes, churning of the topology is averted by means of synchronization. For a SR-node to introduce a new predecessor or leave the ring it has to synchronize with its immediate neighbors, i.e., the predecessor and the successor. Only if they are not in a reconfiguration process themselves a SR-node can either leave or introduce a new predecessor. A SR-node cannot perform reconfiguration operations if any of the adjacent nodes is a reconfiguration process. Hence, restriction of the SR-nodes to their key identifier space partitions in conjunction to synchronization should effectively prevent ring churning. Adjusting the controller to be restrained in performing reconfiguration operations is helpful as well.

Safety-Ring Dynamic Reconfiguration

When it comes to Safety-Ring expansion, each controller triggered SR-node can instruct a regular service instance to join the Safety-Ring. We say that a SR-node promotes an application service instance into the Safety-Ring. In doing so the reconfiguring SR-node can only choose from a set of known service instances that would at the same time become its predecessors if promoted. Since a SR-node is only aware of application service instances that it is monitoring it chooses from them. The set of service instances that are candidates for promotion is denoted $SI_{CND}(si_{sr})$ as:

$$SI_{CND} = \{si \in SI | \exists wi \in WI : si \in wi.SI_a \wedge f_h(wi.id), f_n(si.n) \in KIP(si_{sr}.n\}$$

By ceding a partition of the key identifier space to the new predecessor, the running workflow instances that are mapped into the ceded partition will be reassigned to the new predecessor according to the ring topology self-organization feature. As a result the new predecessor will assume its monitoring and workflow instance storage responsibilities and thus reduce the workload at the successor. Out of the set of candidate nodes the optimal one is chosen by consulting the locally stored extended metadata set on them. The routine `selectOptimal()` implements the multidimensional node ranking of the extended metadata set md_e and returns the node that features the highest rank. Upon promotion, the new SR-node joins the existing Safety-Ring by executing the ring topology joining routine (i.e., Algorithm 3.7) which commences reassignment of Safety-Ring functionalities to it. Algorithm 5.13 shows the Safety-Ring expansion routine.

Note, that Algorithm 5.13 is periodically executed by the controller, i.e., is embedded into it.

When it comes to ring contraction, a SR-node can (instructed by its controller) leave the Safety-Ring so as to conserve local resources at the node. While expanding the ring is as simple as asking a service instance to join, contracting the ring is far more complex. Even though we do not want the controller to be hasty in expanding the ring we assume the addition of SR-nodes to be a welcome occurrence. The reason is that additional SR-nodes should generally result in decreased workload at the others. That is why we choose not to additionally constrain ring expansion routine except for the synchronization with the neighboring SR-nodes.

Algorithm 5.13 $si_{sr}.expandRing()$ Checks the load of the SR-node si_{sr}. In case the global load threshold value tsh_{Lmax} is exceeded the controller is consulted with the local statistic on it $stat(si_{sr})$. It handles the reconfiguration instruction by the controller it selects out of the candidate set $SI_{CND}(si_{sr})$ the optimal and instructs it to join the Safety-Ring.

1: $\exists h(si_{sr}.n, \tau_{max}) \in H(si_{sr}.n) : \nexists h(si_{sr}.n, t_x) \in H(si_{sr}.n)$ **where** $t_x > \tau_{max}$
2: **if** $h(si_{sr}.n, \tau_{max}) \geq tsh_{Lmax}$ **then**
3: $\quad \exists stat(si_{sr}) \in STAT_{sr}$
4: $\quad isOverloaded \leftarrow n.checkFrequency(stat(si_{sr}).f_{er}, stat(si_{sr}).\tau_{er}, \omega_{er}, tsh_{er})$
5: $\quad$ **if** $isOverloaded =$ **true** $\wedge\, conf =$ **false then**
6: $\quad\quad conf \leftarrow$ **true**
7: $\quad\quad n' \leftarrow n.selectOptimal(SI_{CND}(si_{sr}))$
8: $\quad\quad$ **if** $n' \neq$ **null then**
9: $\quad\quad\quad n'.join(n)$
10: $\quad\quad$ **end if**
11: $\quad\quad conf \leftarrow$ **false**
12: $\quad$ **end if**
13: **end if**

The departure of SR-nodes is far more serious as the some consequences of it would not be welcome most of the times. A leaving SR-node has to leave behind the Safety-Ring in a stable situation. Otherwise, its departure could put the remaining into critical situations. For example, if a adjacent node of a leaving SR-nodes is overloaded, the departure of it could put them under additional workload. As sa consequence, one of them could fail due to overload and cause failures of others etc. In face of overladed neighbors a SR-node should better not leave the ring.

Moreover, if the departure of a SR-node cedes a key identifier space partition of significant size to the successor it might also become overloaded. Given a high number of workflow instances that are mapped into the ceded key identifier space partition the additional overhead of their storage might put significant a workload on the successor. The departure of a SR-node can also leave the successor behind with a too big key identifier space partition. This is especially the case for low SR-node configurations. If the key identifier space partition of the successor becomes too big, i.e., grater than $|KI|/r$, then it might become a backup node of itself:

$$|KIP(successor)| \geq \frac{|KI|}{r}. \tag{5.13}$$

Due to the symmetric replication in the system, data items might end up at the same node by applying the *symm* operator if Expression 5.13 applies. Hence, a SR-node should not leave if its successor will end up with a big key identifier space partition.

Finally, if a SR-node is currently monitoring active service instances, the departure of it will leave the service instances unmonitored. Fault handling of such nodes will be unnecessarily delayed until the successor copies the corresponding workflow instances and thus takes over. Hence, a SR-node should not leave in case it is monitoring application service instances.

To cope with the undesired effects of SR-node departure the Safety-Ring is constrained by a set of preconditions that have to be fully met by the SR-node before it can leave. If even a single one is not met, the instructed SR-node cannot leave. The preconditions are summarized as follows:

1. Neither the predecessor nor the successor may be in an overload state or in a configuring state.

2. The yielded key identifier space partition to the successor may not increase its key identifier space by $1/r$ of the whole KI.

3. The leaving SR-node should not monitor any SR-nodes

Only once all three requirements have been met by a SR-node it is allowed to leave the Safety-Ring by means of the ring topology routine `leave()`. Algorithm 5.14 shows the Safety-Ring contraction routine.

Algorithm 5.14 $si_{sr}.contractRing()$ Checks the load of the SR-node si_{sr}. In case the global load threshold value tsh_{Lmax} is not matched the controller is consulted with the local statistic on it $stat(si_{sr})$. If the controller detects underutilization, the adjacent nodes are not overloaded and locked, the resulting partition of the successor is not too big, and si_{sr} is not monitoring any service instances then it makes n leave the ring.

1: $\exists h(si_{sr}.n, \tau_{max}) \in H(si_{sr}.n) : \nexists h(si_{sr}.n, t_x) \in H(si_{sr}.n)$ **where** $t_x > \tau_{max}$
2: $\exists h(predecessor, \tau_{max}) \in H(predecessor) : \nexists h(predecessor, t_x) \in H(predecessor)$ **where** $t_x > \tau_{max}$
3: $\exists h(successor, \tau_{max}) \in H(successor) : \nexists h(successor, t_x) \in H(successor)$ **where** $t_x > \tau_{max}$
4: **if** $h(si_{sr}.n, \tau_{max}) \leq tsh_{Lmin}$ **then**
5: $\exists stat_{sr}(si_{sr}) \in STAT_{sr}$
6: $isIdle \leftarrow n.checkFrequency(stat_{sr}(si_{sr}).f_{cr}, stat_{sr}(si_{sr}).\tau_{cr}, \omega_{cr}, tsh_{cr})$
7: **if** $isIdle =$ **true** $\wedge\, conf =$ **false then**s
8: $conf \leftarrow true$
9: **if** $h(predecessor, \tau_{max}) < tsh_{Lmax} \wedge predecessor.conf =$ **false then**
10: **if** $h(successor, \tau_{max}) < tsh_{Lmax} \wedge successor.conf =$ **false then**
11: **if** $|KIP(n)| + |KIP(successor)| < \frac{|KI|}{r} \wedge Monitoring = \emptyset$ **then**
12: $si_{sr}.n.leave()$
13: **end if**
14: **end if**
15: **end if**
16: $conf \leftarrow false$
17: **end if**
18: **end if**

Note, that Algorithm 5.14 is also periodically executed by the controller, i.e., is embedded into it. Finally, although the SR-node reconfiguration operations should yield a better utilization of the computational resources their excessive application should also

cause inconsistencies in the underlying Finger Tables of the ring topology. The Finger Tables might be inconsistent but the managed data items (i.e., workflow instances) cannot be. Paxos commit transactions prevent writes at inconsistently routed nodes (i.e., learners) till the Finger Tables have converged. Another simple solution to excessive reconfiguration prevention of the ring topology is the proper adjustment of invocation intervals for the contraction and expansion routines. This however is dependent on end-user application of the Safety-Ring.

Dynamic Deployments of Application Service Types

The dynamic reconfiguration of the execution environment, in terms of application service type deployments, should not be limited to the Safety-Ring only. Static configurations of application service types have the same negative effects on the distributed execution of workflow instances as discussed already in Section 1.3. In face of the distributed execution model that is based on late-binding, the execution environment should not feature service types of bottleneck characteristics. That is, there should not be service types that feature too few service instances. In case a high throughput of workflow instances is expected from the system, new service instances that are invoked by them should be added. In case a moderate number of workflow instances is run against the system, underutilized service instances should be removed so as to conserve resources. Simply put, the configuration of the execution environment should linearly scale with the overall workload in the system.

We address the dynamic reconfiguring of the execution environment in a decentralized fashion by enabling each orchestrator node with functionality to instantiate and deactivate application service types at arbitrary nodes. For this purpose we assume that the service types can be instantiated and deactivated dynamically without any unforeseeable consequences or side effects. That is, we limit ourselves to dynamic configuration of discrete service types only. The dynamic instantiation of continuous service types is already handled by the passive-standby technique.

The dynamic reconfiguration of the execution environment builds on the same ideas as dynamic Safety-Ring configuration. In a nutshell, we employ a control loop that monitors the throughput of a workflow instances and in case decrease/increase applies corrective measures – service instance instantiation or deactivation. Along the lines of decentralized reconfiguration at each orchestrator we deploy a throughput controller that with a consistent business logic.

Each controller at an orchestrator keeps track of the throughput by counting the number of late-binding actions for the subsequent service type and triggers its reconfiguration operations by taking into account specified threshold minimum/maximum throughput values and specified time windows. However, we do not just tune the controller to operate on plain late-binding occurrences.

Given the distributed execution model we have to take into account that the monitored throughput at the orchestrator is not the same at all times. Although the control flow is steered with load balancing in mind it might not always produce an optimal outcome. For example, induced by out-of-date metadata at the previous orchestrator nodes the control flow of many workflow instances might end up at the same orchestrator. In

face of suboptimal routing of the control flow, and thus its late-binding overload, the controller should not unnecessarily instantiate/deactivate subsequent service types. To this end, the controller has to take into account the existing environment configuration for each subsequent service type before taking reconfiguration operations.

That is why our controllers keep track of the node ranks, in computing the optimal succeeding service instance, in conjunction to the throughput when performing late-binding of the subsequent service type. By maintaining the rank of the optimal service issuance per activity we gain valuable insight on the execution environment. If the late-binding frequently features low ranked service instances for a longer period of time we broadly conclude the service instances of that type to be overloaded and the distributed execution model to be affected by congestion. On the other hand, the late-binding frequently features high ranked for a longer period of time we broadly conclude the service instances of that type to be underutilized and the execution of workflow instances to be uneconomical, i.e., resource wasting. This implies that the controller maintains the throughput for permanently low ranked and permanently high ranked service types when selecting optimal forwarding nodes (i.e., Algorithm 5.1, line no.11) as well.

The controller configuration thresholds, which are used to steering it, are summarized as follows:

1. $tsh_{Rmin} \in I\!N^{+}$ is the low rank threshold value,
2. $\omega_{lr} \in T$ is the threshold value for the low rank evaluation window,
3. $tsh_{lr} \in I\!N^{+}$ is threshold value for the low rank frequency at which a service instantiation operation should be taken.
4. $tsh_{start} \in I\!N^{+}$ is threshold value for the throughput at which a service instantiation operation should be taken.
5. $tsh_{Rmax} \in I\!N^{+}$ is the low high threshold value,
6. $\omega_{hr} \in T$ is the threshold value for the high rank evaluation window,
7. $tsh_{hr} \in I\!N^{+}$ is threshold value for the high rank frequency at which a service deactivation operation should be taken.
8. $tsh_{stop} \in I\!N^{+}$ is threshold value for the throughput at which a service deactivation operation should be taken.

Similarly to the Safety-Ring controller, the throughput controller needs to be manually tuned by a expert end-user administering (i.e., system administrator) the workflow engine depending the self-configuration needs of the high-level application

To compare the controller threshold values against each orchestrator node locally maintains statistics that reflect its congestion/underload occurrences. The orchestrator rank statistics are formally defined as follows:

Definition 5.9 (Rank statistic – $stat_{rk}$)**.** *Rank statistics locally maintained by the orchestrator si_o for a service type $s \in S$ is defined as tuple $stat_{rk}(si_o, s) = (f_{hr}, \tau_{hr}, f_{lr}, \tau_{lr})$ such that:*

- $f_{hr} \in I\!N^{+}$ *is current high rank counter,*
- $\tau_{hr} \in T$ *is the timestamp of the first high rank occurrence,*
- $f_{lr} \in I\!N^{+}$ *is current high rank counter,*
- $\tau_{lr} \in T$ *is the timestamp of the first high rank occurrence,*

The set of all rank statistics for all orchestrator nodes is defined as $STAT_{rk}$. □

Given the rank statistics, the controller can deduce (by means of routine `checkFrequency()`) congestion and underutilization situations the system and trigger reconfiguration operations. In the scenario of frequently low node ranks for a service type, the controller detecting the congestion can try to improve the situation by instantiating a service type at an underutilized node. In the scenario of frequently high node ranks for a service type, the controller detecting the underutilization can try to improve the situation by deactivating a service type at an underutilized service instance.

In order not to over-provision or under-provision the system with service instances we further constrain the controllers with an average throughput limit per service instance. That is, only if at the same time the controller detects congestion and the average throughput per service instance is greater than the instantiation threshold tsh_{start} a new service type may be instantiated. Similarly, only if at the same time the controller detects underutilization and the average throughput per service instance is lower than the instantiation threshold tsh_{stop} an existing service instance may be deactivated.

Selection of the instantiation node is based on a set of candidate nodes, that is extracted from the local metadata set $MD(si_o)$, such none of them already host the instantiation service type. The instantiation candidate nodes are denoted with the set $InstCand$ as follows:

$$InstCand(si_o, s) \subset N \textbf{ where}$$

$$\forall n \in InstCand(si_o, s) \exists md_e(si_o) \in MD_e(si_o) :$$

$$s \notin md_e(si_o).md(si_o).sub_l.A_{next} \wedge h(n) \in md_e(si_o).md(si_o).\rho_h \wedge (s, n) \notin SI$$

Selection of the deactivation service instance is based on a set of candidate nodes, that is extracted from the local metadata set $MD(si_o)$, such all of them host the deactivation service type. The deactivation candidate nodes are denoted with the set $DeInstCand$ as follows:

$$DeInstCand(si_o, s) \subset N \textbf{ where}$$

$$\forall n \in DeInstCand(si_o, s) \exists md_e(si_o) \in MD_e(si_o) :$$

$$s \in md_e(si_o).md(si_o).sub_l.A_{next} \wedge h(n) \in md_e(si_o).md(si_o).\rho_h \wedge (s, n) \in SI$$

To choose the best candidate node out of the sets $InstCand$ and $DeInstCand$ we rely on the routine `selectOptimal()` that returns the best ranked node based on locally available metadata on them, i.e., MD_e. The routine `selectOptimal()` implements the multidimensional node ranking (i.e., Expression 5.12) and returns the node that features the highest rank. Algorithm 5.15 shows the dynamic service configuration routine.

Algorithm 5.15 $si_o.updateStatistic(s_n, rk)$. In case the given rank rk is lower than the global threshold value tsh_{Rmin}, it updates the statistics $STAT_{rk}$ for the given service type s_n by means of $checkFrequency()$. If s_a is congested and the average throughput per service instance is being higher than tsh_{start} it adds a new service instance to SI. Thereby, out of the set $InstCand(si_o, s)$ the best ranked one is chosen. If rk is bigger than the global threshold value tsh_{Rmax} the statistics $STAT_{rk}$ are updated as well. If s is underutilized and average throughput per service instance is being lower then tsh_{stop} it removes a service instance from SI. Thereby, out of the set $DeInstCand(si_o, s)$ the best ranked one is chosen.

1: $SI_s = \{si \in SI | si_s.s = s_n\}$
2: **if** $rk < tsh_{Rmin}$ **then**
3: $\exists stat_{rk}(si_o, s_n) \in STAT_{rk}$
4: $isCongested \leftarrow n.checkFrequency(stat_{rk}(si_o, s_n).f_{lr}, stat_{rk}(si_o, s_n).\tau_{lr}, \omega_{lr}, tsh_{lr})$
5: **if** $isCongested =$ **true** $\wedge \frac{stat_{rk}(si_o, s_n).f_{lr}}{|SI_s|} > tsh_{start}$ **then**
6: $n' \leftarrow n.selectOptimal(InstCand(si_o, s_n))$
7: $SI \cup (s_n, n')$
8: **end if**
9: **end if**
10: **if** $rk > tsh_{Rmax}$ **then**
11: $\exists stat_{rk}(si_o, s_n) \in STAT_{rk}$
12: $isUnderloaded \leftarrow n.checkFrequency(stat_{rk}(si_o, s_n).f_{hr}, stat_{rk}(si_o, s_n).\tau_{hr}, \omega_{hr}, tsh_{hr})$
13: **if** $isUnderloaded =$ **true** $\wedge \frac{stat_{uis}(si_o, s_n).f_{hr}}{|SI_s|} < tsh_{stop}$ **then**
14: $n' \leftarrow n.selectOptimal(DeInstCand(si_o, s_n))$
15: $SI \setminus (s_n, n')$
16: **end if**
17: **end if**

Deactivating service instances is not always as straightforward as instantiating them. Although a service instance has been chosen for deactivation due to a very high rank and negligible average workflow instance throughput it might be in the process of serving an invocation request. When deactivated, a regular workflow instance execution is artificially terminated for the purpose of economical resource consumption. In case the execution time of such a service type is long, or costly, due to some other execution aspect, the forced deactivation of it might turn out to be not so beneficial after all. To overcome this the deactivation routine could be additionally constrained with an allowed number of currently invoked service instances or even with the semantical importance of the service type. In any case, the expert end-user should always choose to set all threshold values to be moderate in terms of deactivation, and lavish in terms of instantiation. In general, a forcefully deactivated running service instance will not affect correctness of it as it will be detected by the Safety-Ring as a failure and recovered immediately at a backup node.

When it comes to decentralized reconfiguration of the environment, our approach is conditioned on the freshness of the locally available extended metadata set MD_e. If metadata changes are immediately visible at the subscribed orchestrator nodes more

accurate decisions can be made when selecting the optimal reconfiguration node. This even true when redundancy of the metadata at different orchestrator nodes is considered. If all redundant orchestrator nodes feature locally the same metadata they will concurrently come to the same optimal node decisions in applying the same business logic, i.e., the same routine `selectOptimal()`. Formally, the same optimal reconfiguration node will be selected in a distributed fashion if:

$$si_{o1}.selectOptimal(DeInstCand(si_{o1}, s)) = si_{o2}.selectOptimal(DeInstCand(si_{o2}, s)) \implies$$

$$DeInstCand(si_{o1}, s) = DeInstCand(si_{o2}, s) \wedge$$

$$\forall n \in DeInstCand(si_{o1}, s) \exists md_e(si_{o1}) \in MD_e \wedge \exists md_e(si_{o2}) \in MD_e :$$

$$h_{o1}(n,t) \in md_e(si_{o1}).md(si_{o1}).\rho_h \wedge h_{o2}(n,t) \in md_e(si_{o2}).md(si_{o2}).\rho_h \wedge$$

$$m_{o1}(n,t) \in md_e(si_{o1}).\rho_m \wedge m_{o2}(n,t) \in md_e(si_{o2}).\rho_m \wedge$$

$$st_{o1}(n,t) \in md_e(si_{o1}).\rho_{st} \wedge st_{o2}(n,t) \in md_e(si_{o2}).\rho_{st} \wedge$$

$$lt_{o1}(n,t) \in md_e(si_{o1}).\rho_{lt} \wedge lt_{o2}(n,t) \in md_e(si_{o2}).\rho_{lt}$$

The same constraint applies for the activation candidates set *InstCand* as well. However, any two orchestrator nodes are more likely to differ in the sets *InstCand* than in the sets *DeInstCand* which are always the same for all. This can be attributed to the existence of subscriptions at them. For instance, if a orchestrator hosts more than one application service type, it will feature for each one of them subscriptions to the subsequent service types. As a consequence, the activation candidates set *InstCand* should feature more nodes the more service types are hosted at a node. Thus orchestrator nodes are not as likely to derive the same optimal instantiation nodes as they are in case of deactivation nodes.

Our approach to dynamic reconfiguration of the environment is influenced by the state of the centralized metadata repository which supplies the decentralized controllers with metadata. In case metadata repository features overload situations the freshness of metadata should differ at the controllers and thus the candidate node sets. However, given the freshest metadata at the controllers, the instantiation/deactivation decisions will be concentrated to a limited set of optimal nodes and thus should yield at runtime addition/removal of service instances which is gradual at and not widespread. Hence, by keeping the metadata repository working at moderate capacity the decentralized reconfiguration approach should deliver the expected results.

5.3 Summary

In this chapter we have described how the distributed execution model is enhanced in a novel way for reliability and performance without affecting any of its main characteristics such as scalability.

First, we have concluded that the distributed execution model is vulnerable to failures of active service instances. To overcome these we have introduced the Safety-Ring system service. SR-nodes perform monitoring of active service instances and handle

their failure with a late-binding of a replacement service instance. To posses the data for late-binding purposes the SR-nodes organize themselves into a reliable, scalable and consistent data store for storage of workflow instances. That is, the SR-node from a Chord ring that supports symmetric replication and Paxos distributed transactions. The benefit of such an approach is that the SR-nodes are reliable and scalable themselves and so is the Safety-Ring service. Hence, scalability of the distributed execution model is preserved by redirecting the late-binding step through the scalable Safety-Ring with a successful write of the workflow instance. However, the price of reliability of the distributed execution model is reduced throughput performance.

To extend the Safety-Ring service to heterogeneous execution environments in this chapter we introduce the Compass data access protocol. Compass complements the Chord ring of Safety-Ring with node latency information. This way Compass can compute the latency optimal paths to each node in the Chord ring and use when needed instead of the Finger Tables. By taking the latency optimal paths more forwarding steps can be taken as compared to Chord. The latency optimal paths are maintained inside Compass table that are of linear size relative to the number of nodes in the Chord ring. The Compass Table can however be downsized at the price of optimal path precision.

This chapter additionally introduces novel approaches to continuous data flow reliability. While the reliability of the control flow is guaranteed by the Safety-Ring, continuous data flows (i.e., data streams) necessitate more resource friendly recovery mechanisms. In this chapter we have extended the traditional passive-standby approach for redundancy. Namely, traditional passive-standby, features periodic checkpointing of streaming instance state to one backup node only. By increasing the number of backup nodes we have improved the robustness of continuous data flows to node failures. To enforce the consistency of the redundant backups we have applied the 2PC protocol for the coordination of the asynchronous data replication processes. The 2PC protocol has even be modified such that it is not blocking in the event of a coordinator failure. The price for the improved reliability of continuous data flows is slightly increased network overhead (i.e., additional backups) per active steaming instance, and potentially longer recovery times in case of node failures.

With the Safety-Ring and redundant passive-standby extensions we have shown in this chapter how the distributed execution model has been adapted for reliability. This chapter addressed also the problem of suboptimal (static) service instance configurations in the environment in terms of workflow instance throughput. The approach of this chapter is based on dynamic deployment of service instances at arbitrary nodes. In this chapter, dynamic reconfiguration of the environment is achieved in a decentralized fashion by means of distributed controllers that perform autonomous instantiation/deactivation of service instances. To this end they rely on accurate metadata on the environment and local statistics. To capture additional aspects of dynamic behavior inside the heterogeneous execution environment we have introduced the stability, mobility and latency metadata next to workload. The additional metadata is maintained by the global repository and published to all subscribed controllers, which in turn use the metadata for optimal deployment node detection. In instantiating/deactivation service instances, the controllers are however limited to application service types (of discrete nature) and SR-nodes only.

6

Implementation

In this chapter we describe the software implementation of the distributed workflow execution model introduced in the previous chapters. We start this chapter with the description of the OSIRIS framework that will serve as the basis for our work as introduced in Chapter 5. OSIRIS [SWSjS04, SST+05, STS+06] is a JAVA based framework for the Peer-to-Peer execution of distributed processes, that address the workflow management concepts of Chapter 4 to the biggest possible extent. In terms of OSIRIS, we provide an architectural overview of its software stack that is inter-operable with any JAVA execution environment. Further, we introduce all of its extensions – OSIRIS-SE [BS07, BS11a, BSS05, Bre08], OSIRIS-ON [MS10b, Moe12] and OSIRIS-PRO [Zel11] designed for the improvement of various aspects of distributed workflow management. Finally, we conclude this chapter with the elaboration of the integration process of our work into the OSIRIS framework.

6.1 The Workflow Engine Implementation

The OSIRIS framework implements the distributed process execution along the lines of the concepts found in Chapter 4. The only difference is that OSIRIS operates on distributed processes or even composite services, which can be from a semantic point of equivalent to our workflows. That is, both distributed processes and composite services structure a set of atomic activities into a partial execution order.

In correspondence to the distributed workflow execution model, OSIRIS features the necessary system services,i.e., the orchestration service and the repository service. That is, each OSIRIS peer is equipped with an orchestration service and is all times subscribed to the repository service. In turn, the repository service is usually maintained at one peer only, i.e., the super peer. The control flow along with its corresponding workflow instance is embodied with of a token data structure that is referred to as the *Whiteboard*. The Whiteboard token is the subject of exchange among the peers, and the possession over it puts the orchestrator node in control of the workflow instance. Only once in possession of the token the orchestrator can perform invocations of application service instances. The other ones have to wait for their turn. The data flow is

also maintained within the Whiteboard control token, but only if the service types are of discrete nature. Orchestrator nodes in control of the token map the conveyed data into invocation request parameters and store the resulting values in the token for subsequent service instance invocation purposes. Continuous data flows are implemented by streaming the data directly among the activated streaming instances. Therefore, the distributed execution model is implemented for the control flow and the discrete data flow by means of the Whiteboard control token which is subject to collaborative migration among the orchestrator peers.

6.2 The OSIRIS Framework

OSIRIS stands for *Open Service Infrastructure for Reliable and Integrated process Support* and is in its essence a message-oriented middleware (MOM) that supports a modular functionality architecture. The foundation of the OSIRIS framework are the components that feature any system or application service specific functionality. The components are loosely coupled by means of the messaging middleware so as to form a coherent functional whole. This allows for dynamic functionality adaptation, as components can be activated/deactivated on the fly. Thereby, the message exchange among components can even span remote computational devices

Per default OSIRIS provides a rich set of components, that provide basic workflow management functionality, and that are available to any application service. For instance, the orchestration system service is an OSIRIS component as well as the repository system service is. In turn, application services can either be implemented by OSIRIS components (manually) or can feature external Web Services which are again accessed through endpoint components (e.g., SOAP endpoints) of OSIRIS.

An OSIRIS component is designed to be lightweight, in terms of memory and CPU requirements, so that general purpose deployments to any kind of computational environment are feasible. To exploit the locally available hardware resources in a scalable fashion (in particularly the CPU core numbers) for providing a high performance functionality execution of a component, OSIRIS exploits the Actor computation model. Precisely, the business logic of a component is embedded into the messaging middleware with of a set of stand-alone tasks that are concurrently executed by a set of *Actors*. Each tasks directly implements a part of the businesses logic in the from of a message handler that maps to a specific message type. Upon reception of an instance of a concrete message type the corresponding handler is found and then executed by the an actor. Thereby, the incoming messages, stemming from other components, are first placed in a message queue and then handled by the first available Actor in line with the corresponding handler. OSIRIS implements the message queue with a Java first-in-first-out (FIFO) dequeue and each Actor with of a Java thread. Per default, OSIRIS assigns to each message queue a number of Actors which corresponds to the number of available CPU cores at the peer. The number of Actors can be dynamically increased/decreased at runtime. The Actors of the system components are assigned with the highest thread priority whereas the Actors of the application components are assigned with the default thread priority. To power the message handler embedded business logic a compo-

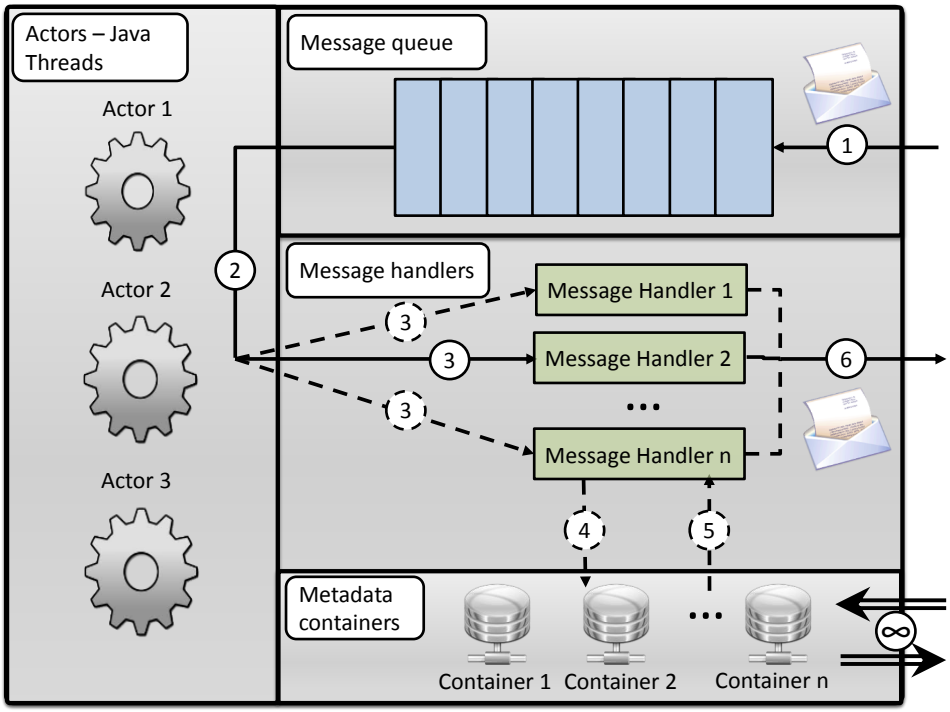


Figure 6.1: OSIRIS component architecture.

nent can feature a set of data store containers. Each container is founded on the publish/subscribe message exchange pattern so as to replicate metadata among the components. The component architecture of OSIRIS is depicted in Figure 6.1.

The Actor based message processing of an OSIRIS component is depicted in Figure 6.1. As the figure shows, the incoming message is firstly (i.e., step no.1) enqueued at the message queue (i.e., depicted with the blue rectangles). Afterwards, the first available Actor (e.g., Actor no.2, depicted with the gear-wheel) dequeues the message (step no.2) and maps the message to corresponding message handler (e.g., Message Handler no.2, depicted with the green rectangle). At the next stage of the processing (step no.3), the encoded business logic of the message handler is executed by the Actor. In case the message was sent to the component in a pub/sub container (i.e., depicted with the silver cylinders) related context, the message hander execution involves the update or querying of some container. That is, the data conveyed by the message is to be stored at some container (e.g., step no.4), or data is to be read from some container (e.g., step no.5) so as to complete the execution of the message handler. Where necessary an output message is produced and sent out to some other component (i.e., step no.6) as the result of the message handler execution.

The message middleware functionality, that is responsible of connecting all components, is implemented with a special purpose component that is referred to as the *Horus* component. The Horus component serves as a peer local message routing hub that accepts all incoming messages and redirects them to the appropriate component. All message communication among the components has to necessarily go through the Horus. In order to be able to know the accurate component destination the Horus relies

on a container that stores node addresses and a container that stores node component occurrences. The repository that features the occurrence of components at nodes essentially corresponds to the local metadata of the service instance repository. Moreover, the Horus is capable of dynamic component activation/deactivation at runtime. A message is essentially implemented also as a serializable key-value associative array.

The Whiteboard itself is a specific implementation of an OSIRIS message, i.e., a hashtable. At runtime there might exists multiple copies of a Whiteboard for the same workflow instance. Multiple copies are created whenever a forking activity is migrated (e.g., for each parallel activity one Whiteboard is created) which are all merged into one copy at some join activity.

In general, the global metadata repository as elaborated in Section 4.2.2 (i.e., Algorithms 4.3, 4.4) is implemented with the *Repository* component that all other containers of all the other components are subscribed to for pub/sub metadata replication. The repository component is however usually hosted at one node only, i.e., at the *super-peer*. This does not have to be always the case as the global repository can be horizontally distributed to an arbitrary number of nodes. The Repository component can be partitioned along the separate metadata sets. The only prerequisite is that all nodes know about the specific locations of the distributed repository components. The Repository connects all peers by locally accepting and managing all subscriptions. Updates to state at peers are first issued to the repository where they are locally collected (i.e., updated) upon which they are subsequently propagated to all other peers that are determined based on subscriptions and affected by the updates. Hence, the repository accumulates an aspect (e.g., workload) of the execution environment metadata and efficiently publishes updates parts of it to subscribed peers. Each subscription to the repository is further associated with a set of predicates that facilitate the reduction of replication data volumes. For instance, the subscribers can specify the relative change (e.g., 10% in the change of workload), in data before it becomes a subject of propagation to them. In case of continuous and frequently changing (e.g., workload) discretization predicates can be applied so as to map the data to a finite set of discrete values. Freshness predicates (i.e., *eager* and *lazy*) are also applicable. Subscriptions that feature a *lazy* freshness predicate are subject to batching, compression of multiple data items and data propagation in quiesced state of the system.

The control flow based migration of workflow instances (i.e., the Whiteboards) is implemented with the *Orchestrator* component. Next to node component container the Orchestrator component additionally relies on a container which features local metadata, in terms of node workload. The amount of node metadata to be maintained is naturally determined by the locally available application components and their control flow dependencies to other ones.

OSIRIS does not only provide pub/sub metadata replication and Whiteboard migration, which jointly yield a peer-to-peer workflow instance execution. Per default, OSIRIS also offers of a rich set of distributed workflow management features that have been added to it over the years. The most important features among many others include continuous data flow management, semantic execution models and intuitive workflow definition description etc. The OSIRIS extensions can be best summarized as follows:

- OSIRIS-SE. *OSIRIS Streaming Enabled* enforces the reliability of continuous data flows by means of the passive-standby approach as introduced in Section 4.1.2. To this end it introduces a *Stream* component that in enhanced with (input/output) data item buffers so as to handle the data streams (i.e., reliable data item sending, ordering of data items etc.) and that implements the algorithms 4.1 and 4.2. Moreover, OSIRIS-SE additionally features algorithms that improve checkpoint efficiency for resource limited nodes, i.e., the *Coordinated Checkpointing (COC)* and the *Efficient Coordinated Checkpointing (ECOC)* algorithm. ECOC features coordination of checkpoints among all downstream service instances which allows for the omission of the input/output buffers from the checkpoint state. Being the biggest portion of a checkpoint data payload the omission of buffers yields a significantly reduced resource consumption of bandwidth and data storage per node.

- OSIRIS-ON. *OSIRIS Next* enforces the reliability the of control flow by means of semantic services as introduced in the advanced execution models of Section 4.1.2. OSIRIS-ON recovers failures of service instances at runtime by means of alternative yet semantically equivalent activity execution paths. To represent service semantics the symbolic approach based on Description Logics is leveraged. Thereby, the alternative activity paths are not predefined but are resolved at recovery time in a forward-oriented and optimistic fashion while preserving the correctness of execution. To this end OSIRIS-PRO introduces the *ON* component, that features a knowledge base for the semantic description of service type functionality. For instance their preconditions and effects etc. Moreover, the ON component features a set of novel algorithms that allow for the semantic detection of failures and the dynamic adaptation of the control flow at runtime.

- OSIRIS-PRO. *OSIRIS Process* applies the *programming in the large* [DK76] software development approach to the description of workflow definitions[1]. OSIRIS-PRO introduces a novel programming language that closely resembles, in terms of intuitiveness and mightiness, to the well known programming language JavaScript [GM04]. The novel programming language is powered by the *Compiler* component that embeds the Scala [Oa04] programming language interpreter into it. The embedded interpreter allows for just-in-time interpretation of workflow definition code at runtime and thus dynamic insertion of functionality (i.e., code) into the components. Hence, OSIRIS-PRO enables the system for dynamic functionality change of components. Moreover, it adds to workflow dentitions other programming language constructs such as pointers and exceptions etc. Essentially, these map under the hood to underlying data store and failure recovery facilities such as the Safety-Ring storage and recovery.

All available components of OSIRIS are wrapped into a thin software stack that is referred to as the OSIRIS layer and that can be deployed at any peer running the Java runtime environment (JRE). Thereby, the system components are always lying on top of the system components and are combined with them (via the Horus messaging) so

[1] In OSIRIS-PRO terminology, a workflow definition is commonly referred to as a process

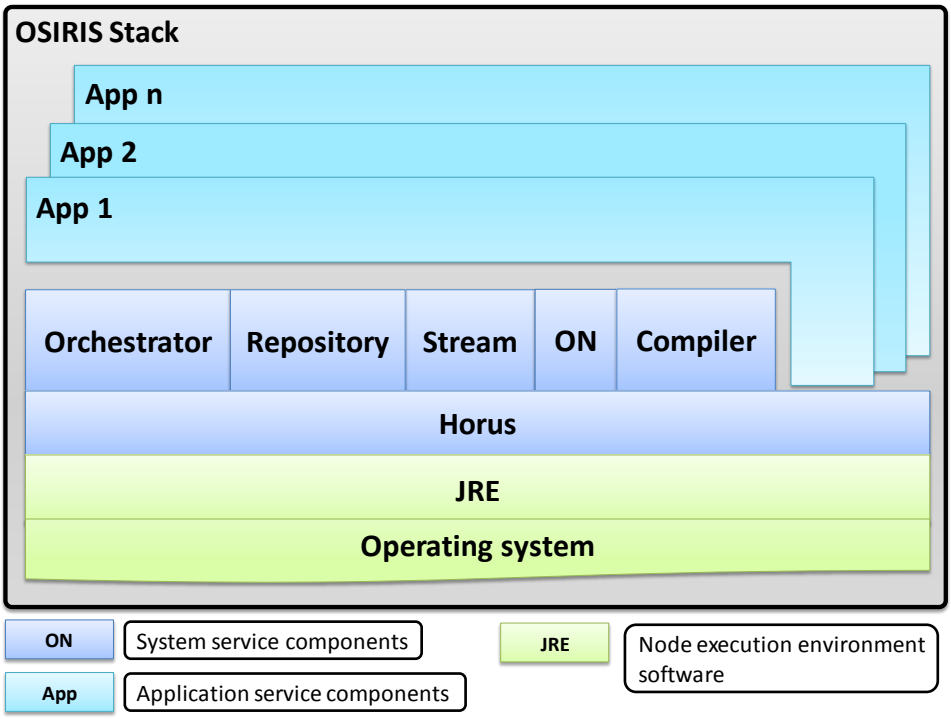


Figure 6.2: OSIRIS layer architecture.

as to support the desired functionality of the application service. Figure 6.2 depicts the OSIRIS layer architecture.

For instance, if OSIRIS is to be run on resource limited (e.g., mobile) devices or commodity hardware, minimalistic variants that contain only the most necessary components of the OSIRIS layer can be used. If OSIRIS is to be hosted at powerful computational cluster nodes or Cloud environment instances, the full fledged OSIRIS layer containing all the existing components can be deployed. Due to the high modularity of the its component stack, OSIRIS is open for virtually unlimited application possibilities. That is, depending on the concrete application scenario the right combination of components merely has to be selected.

6.3 OSIRIS Safety-Ring

In order to implement the advanced concepts of distributed workflow management, as introduced in Chapter 5, the default OSIRIS framework is extended with a new set of features. Precisely, the OSIRIS layer is extended with a new set of system components that put into effect the self-healing and self-optimizing distributed workflow execution model. These features of the system are embodied with the *OSIRIS-SR* [PSH13, SS13, SPS13, SS12, SH12, MS10a], *OSIRIS-rSE* and *OSIRIS-PROs* extensions of OSIRIS. While OSIRIS-SR and OSIRIS-rSE deal with reliability of execution of the control flow and the data flow, OSIRIS-PROs focuses on service type dynamic hot-deployment. The exact

contribution of each OSIRIS extension as well as its functional dependency to others is summarized as follows:

- OSIRIS-SR – The reliable control flow execution feature is implemented with the OSIRIS-SR extension. *OSIRIS Safety-Ring* implements as the name already suggests the Safety-Ring concept of Section 5.1.1. Since Safety-Ring is dependent on a number of fundamental distributed data management concepts OSIRIS-SR implements a number of general purpose system components so that they can be utilized by any other component as well. As a basis for further extension OSIRIS-SR introduces the *Paxos* and the *Chord* component. The Paxos component implements the reliable distributed transaction concept of Section 3.3.2 along with all of its algorithms, i.e., Algorithm 3.18 – Algorithm 3.32. In line with the concepts of a symmetric replication based Chord distributed hash table of Section 3.1 and Section 3.2 the fundamental *Chord* component is introduced by OSIRIS-SR as well. Thereby, the Chord component implements the algorithms 3.1 – 3.7 and algorithms 3.10 – 3.14. To improve the performance of Chord, in terms of data lookup efficiency, for heterogeneous node environments OSIRIS-SR introduces the *Compass* component. The concepts of latency optimal data access, as introduced in Section 5.1.2, are carried out by of the Compass component and its implemented algorithms 5.4 – 5.7. The Compass component builds upon the Chord component and cannot function without an underlying Chord ring. Finally, the *Safety-Ring* component implements the reliable distributed execution model by applying the Paxos commit on top of Chord (i.e., Algorithms 3.33 – 3.40) so as to create a scalable, reliable and consistent data store. The Safety-Ring data store is then merged with the control flow by implementing the algorithms 5.1 – 5.3 so as to ensure it for failures.

- OSIRIS-rSE – The reliable data flow execution feature of OSIRIS-SE is enhanced for redundancy by means of the OSIRIS-rSE extension. *OSIRIS redundantly Streaming Enabled* implements the redundant passive-standby recovery strategy for continuous data flows as introduced in section Section 5.1.3. For this purpose the 2PC distributed transaction protocol of Section 3.3.2 is needed so as to guarantee consistency of checkpointed data. OSIRIS-rSE implements the 2PC protocol (i.e., algorithms 3.15 – 3.17) with the general purpose system component that goes by the same name. The *2PC* component is open for usage to any other component of the OSIRIS layer. The redundant passive-standby mechanism is embodied with the joining the 2PC component and the Streaming component inside the correspondingly named system component. To this end, the *Redundant Passive-standby* component implements Algorithm 5.8 along with Algorithm 5.10.

- OSIRIS-PROs – The service instance hot-deployment concept, aiming to improve throughput of workflow instances, is implemented with the OSIRIS-PROs extension. Thereby, *OSIRIS Process Services* bases its implementation on the distinction between application service types and system service types just as elaborated in Section 5.2.2. Thereby, application service types are dynamically deployed by means og the *Service Balancer* component such that it implements the Algorithm 5.15. Although the Service Balancer component can be used by any other

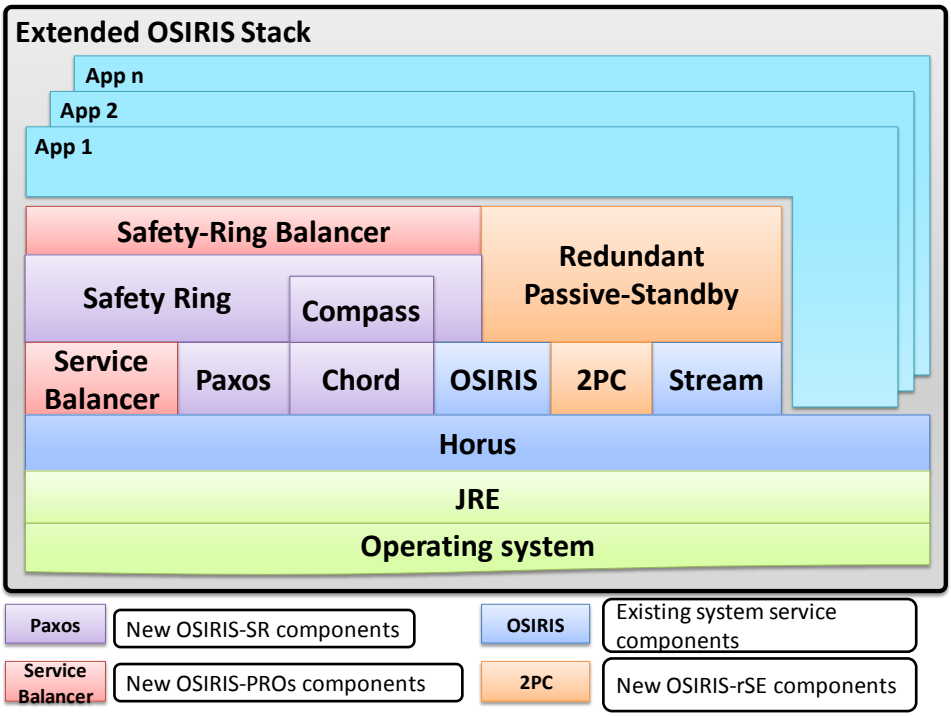


Figure 6.3: Extended OSIRIS layer architecture.

one of the OSIRIS stack, it is also an important building block for the reliable control flow of the Safety-Ring component. That is, only the Safety-Ring improves the workflow instance throughput with the help of the Service Balancer component. In terms of system services, only the Safety-Ring has been enabled for dynamic deployment. To this end, the component *Safety-Ring Balancer* implements the ring expansion and contraction algorithms 5.13 – 5.14 with the Safety-Ring component as its building block. However, the OSIRIS-PROs extension does not only add new components to the stack. It also modifies the existing ones. Precisely, the Repository component is extended for the additional metadata on nodes such as stability, mobility and latency, as elaborated in Section 5.2.1.

An overview of the extended OSIRIS layer with the new components and their functional hierarchies is shown in Figure 3.15. Note, that although the extended OSIRIS layer features a significant number of new components, and thus an increased memory and CPU footprint, it is very unlikely that all of them will be deployed at the same time. Precisely, we assume that Safety-Ring nodes cannot be active streaming instances at the same time in practice, i.e., OSIRIS-SR components and OSIRIS-rSE cannot be deployed simultaneously. Whereas combinations of components of OSIRIS-SR and OSIRIS-rSE with OSIRIS-PROs are always possible.

Finally, Figure 6.4 illustrates the big picture of the OSIRIS based workflow execution. As the picture shows OSIRIS fully incorporates the distributed workflow execution model as discussed in Section 4.2 and in the example pictures Figure 4.3 and Figure 4.4.

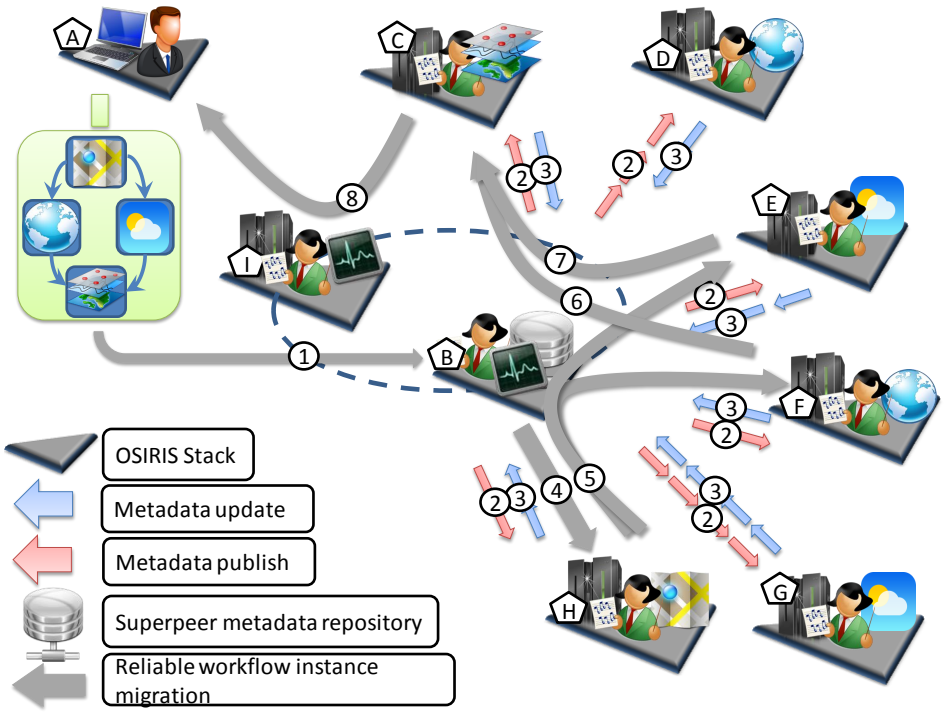


Figure 6.4: The OSIRIS execution big picture.

Example 6.1

As illustrated in Figure 6.4 all peers now commonly feature the OSRIS stack, that encompasses the application and system services. This means that even the super peer (i.e., peer *B*), which usually only hosts the metadata repository, can also participate to the construction of the Safety-Ring itself. All the steps of the distributed execution model are followed through, except that instead of the workflow instance the Whiteboard is migrated among the peers. Whenever a workflow instance enters the system it is first sent to the super peer upon which metadata on it is propagated to all peers of interest – steps 2 - 3. Afterwards, based on control flow precedence order the late-binding of peers with the Whiteboard is performed in a reliable fashion – steps 4 – 8. That is, each step between 4 and 8 is considered to be reliable as it goes through the Safety-Ring. In contrast to the traditional distributed execution model, the join repository is circumvented, in letting a corresponding SR-node perform the joining. This is however, inherent to the reliable distributed execution model of Section 5.1.1.

7

Evaluation

In this chapter, we provide experimental results that validate the concepts of our work. That is, we evaluate the self-healing and self-optimizing features of our work aimed at improving the distributed workflow execution model. To conduct the experiments we first need to define a common execution environment and notation that will be used throughout all experiments. Then we introduce the baseline system that represents the current state of distributed workflow management. Once this is established our contributions can be added and evaluated against the base line.

To evaluate the improved reliability of Safety-Ring we will run inside the execution environment a number of workflow instances that are subjected to random node failures. Thereby, we will evaluate the Safety-Ring in different SR-node configurations so as to examine the effects of Safety-Ring on the scalability. Precisely, we will investigate whether the reliability of Safety-Ring has an affect on the scalability of the distributed execution model. We will measure the effects of reliability with the throughput of workflow instances. In theory, with the increase of SR-node numbers the throughput of workflow instances should be higher and node failures should be recovered faster yielding again more processed workflow instances. On the contrary, with no Safety-Ring around, workflow instances should start disappearing. We will increase the size of the execution environment as well so as to follow up on the scalability investigation. With more nodes in the environment the Safety-Ring should in theory perform even better and facilitate throughput due to its scalable architecture.

To evaluate Safety-Ring for node heterogeneity we will introduce nodes of different hardware characteristics (i.e., CPU and network bandwidth) and of different numbers inside the environment. We will also upscale the heterogeneous execution environment so as to investigate scalability of Safety-Ring w.r.t. heterogeneity of nodes. The Compass extension of Safety-Ring should help to improve throughput of workflow instances in heterogeneous settings even for the upscaled configurations.

Once, Safety-Ring has been evaluated for reliability and scalability we will evaluate the workflow engine w.r.t. continuous data flows, i.e., data streams. We use a different execution environment that is comprised of resource limited nodes only. In this environment we will create a continuous data flow that is subjected to node failures. The majority of nodes will be faulty. Not all of them will actively participate to the stream-

ing of data as some will serve as backup nodes. By increasing the number of backup nodes in separate experiments we will investigate whether redundant passive-standby improves the robustness of the continuous data flow to node failures. Recap, the more backup nodes are featured by redundant passive-standby, the more robust the continuous data flow should be. We will measure the effects of reliability with the throughput of data. More reliable settings should yield more transmitted data, as less data items should get lost due to failure.

Finally, we will evaluate the self-optimization feature of the distributed execution model. Thereby, we will limit ourselves to the dynamic deployment of application service instances only. For these evaluations the same heterogeneous environment will be used as for the Safety-Ring evaluations. Thereby, we will also upscale the heterogeneous environment so as to investigate the effects of scalability on our decentralized approach to dynamic service deployment. Recap, with more nodes at disposal the controllers should have more variety in selecting optimal deployment nodes. We will measure the effects of self-optimization with the throughput of workflow instances.

7.1 Basic Experimental Setting

The purpose of this section is define common execution environments that will be used for the evaluation of the self-healing and self-optimization features. To do so we characterize the execution environment by the number of all participating nodes, the number of mobile nodes, and the number of faulty nodes. Against this configurable execution environment a common workflow definition is used for all experiments.

Since our work aims at improving the reliability of the distributed workflow execution disjointly in the context of the control flow and the continuous data flow (data streams) we separate our experiments with that regard. For the purpose of evaluating the contributions of the reliable control flow execution we devise a specific system setting that is composed of a specific execution environment and a specific workflow definition. Likewise for evaluating the contributions of reliable data flow of continuous type we devise a specific experimental setting as well.

7.1.1 Environment for the Control Flow Evaluations

To evaluate the control flow for its dynamic runtime execution behavior we confine all our experiments to a common workflow definition, a common execution environment and a common experimental procedure.

Workflow Definition for the Control Flow Evaluations

The sample workflow definition we use stems from the dynamic emergency management application scenario of Section 2.3 by mimicking the Disaster Management workflow. Figure 7.1 depicts the used workflow definition for the control flow in BPMN notation, that is composed of eight discrete application services (i.e., A, B, C, D, E, F, G and H) of different type. Each one of them simulates computations at which some of them

are more intensive than others. Two classes of activities with respect to their utilization of CPU device resources are defined as follows:

- *Intensive*: Services of this class are significantly more CPU intensive than the activities of the other class since they are performing heavier, longer computations. That is, each invocation of these activities results series of numerical calculations that simulate real workload and thus create a high CPU load on the hosting node. The services B, F and G belong to this class.

- *Non-Intensive*: Services of this class are not as computationally intensive since they bring less work on the node. That is, each invocation of these services result in simpler numerical computations that are approximately three times smaller than the intensive service ones, in terms of their relative execution time with respect to the underlying computational hardware. The remaining services A, C, D, E and H belong to this class.

Each application service type is depicted by means of Figure 7.1. Thereby, the computationally intensive services are depicted with bold letters and violet colors, whereas the less intensive ones are depicted with blue colors. The orchestration system service, in charge of enacting the workflow definition, is depicted with the green BPMN swimming lane.

Note, that our application services do not involve any meaningful data creation. Precisely, none of the specified application service produces any meaningful and voluminous data that is shipped along with the workflow instance. Neither is the data that is produced, i.e., intermediate computational results of the numerical calculations, written to disk. It is on only kept in main memory during execution. The reason behind that lies in the fact that data transmission among services is not an optimization metric for the control flow. As consequence, this workflow definition neglects the flow of voluminous data among application services and is solely focused on the flow of control and metadata among system services, needed to power them. It is important to mention that voluminous data items that are shipped among discrete application services would not make any significant difference. In OSIRIS-PRO voluminous data items can be offloaded to internal storage and references to them used instead inside the shipped Whiteboards.

Application services that can be invoked in parallel exist within the used workflow definition, as well. These services are structured into parallel branches that are spawned by service B and have to be joined at service G. The first parallel branch is composed of the service C, D and the second of services E, F. The service type D is a primary activity, whereas F is the secondary activity.

To achieve a true control flow based execution, we spread the services across all nodes such that no node can perform multiple subsequent activities locally in deploying only one service per node. Later, with the activation of self-optimizing execution features this state is subject to change and arbitrary number of services might be dynamically deployed at a node. The initial distribution of the first 8 services has been has been randomly determined and then consequently applied on the other existing nodes by means of the modulo operator:

$$Peer_{id} \textbf{ mod } |Services|$$

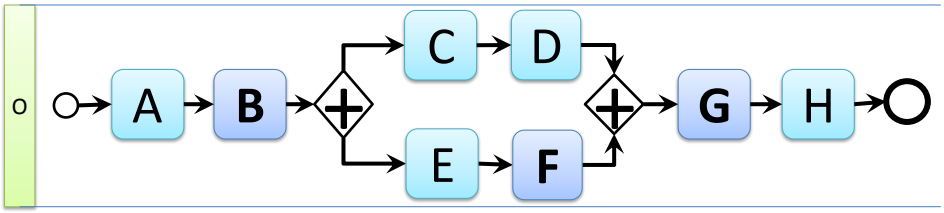


Figure 7.1: The workflow definition in BPMN notation for the evaluation of the control flow.

Moreover, all nodes also feature the orchestration service and all the corresponding OSIRIS components. However, only one of the evaluation nodes is the super-peer and hosts all necessary global metadata repositories (i.e., the Repository component) whereas the other ones are merely subscribers to it.The resulting deployment of services for all nodes is summarized by Table 7.1 as follows:

Table 7.1: Services distribution at nodes

Services	Peers
$Super - peer$	Peer 0
A	Peer 8, Peer 16
B	Peer 7, Peer 15
C	Peer 2, Peer 10, Peer 18
D	Peer 1, Peer 9, Peer 17
E	Peer 3, Peer 11, Peer 19
F	Peer 5, Peer 13
G	Peer 6, Peer 14
H	Peer 4, Peer 12

The distribution of other system critical services such as the Safety-Ring functionality is evaluation dependent and shall be discussed in detail in the corresponding evaluation subsections.

Execution Environment

The execution environment we have chosen to run our experiments against is comprised of two configurations with 11 or 20 nodes, respectively. In terms of hardware characteristics, each of the evaluation nodes corresponds to an Amazon EC2 (Elastic Compute Cloud) virtual device instance. The instance type used in our experiments is c1.medium[1]. Note, that the application of the same AWS instance type for all experimental nodes guarantees equality of hardware characteristics. This way the self-optimizing and self-healing features of our work can be elicited since there are no over-powerful nodes to drastically affect the behavior of the system and suppress the effects of concepts. However, to simulate heterogeneity and volatility inside the computational

[1]http://aws.amazon.com/ec2/previous-generation/

environment, such as in the case of our sample high-level application scenario, we introduce different classes of evaluation nodes:

- *Static*. Static nodes are the plain EC2 virtual instances, which feature moderate compute (2 virtual CPU cores), memory (1,7 GiB) and moderate network performance characteristics (around 100kB/sec).

- *Mobile*. Mobile nodes are simulated by artificially decreasing the network transmission performance of static nodes. That is, just before a message is sent to the network socket dummy data of notable size is attached to it such that is significantly throttles down the actual transmission capabilities of the node. The dummy data corresponds to string data structures that are containing randomly generated number sequences of variable byte size. The precise dummy data size values are determined by the physical distance towards the destination nodes. Since node physical proximity is always subject to random change, the dummy data size values will constantly fluctuate within the limits of 0 - 100kB/sec ms as compared to the static nodes.

- *Faulty*. Faulty nodes are simulated by stopping the OSIRIS processes at the static nodes at a random point in time. The precise stopping moment is determined by a uniformly distributed random function with that remains within the limits of 0 - 320 seconds. Each of the stopped nodes eventually rejoins the execution environment after 60 seconds of waiting time and is again subject to a new random stop.

Heterogeneity and volatility of the execution environment is thus achieved by configuring in separate evaluation runs a certain amount of static nodes to be faulty and/or mobile. Precisely, a relative proportion of 0%, 25%, 50% and 80% out of all static nodes can be configured to be mobile and/or faulty. Thereby, only the nodes $6-11$ (i.e., for the 11 node configuration case), and $10-20$ respectively (i.e., for the 20 node configuration case), will be assigned to be faulty, whereas nodes $3-8$ (i.e., for the 11 node configuration case), and $5-15$ respectively (i.e., for the 20 node configuration case) will be assigned to be mobile. The detailed node class configurations of the are evaluation dependent and shall be discussed in detail in the corresponding evaluation sub-chapters.

Experimental Procedure

The simple composition of the evaluation workflows is designed to correspond to a short running execution request issued by a system user for each experiment. Hence, user requests are mimicked with a queuing model by triggering an instance launch every t milliseconds. Thereby, the super-peer will solely serve as a measurement and workflow instance launching node. To elicit the self-optimization features of the system we will create a significant load on the system, in terms of running workflow instances, by decreasing t by 15% every minute. For each experiment, we will set the initial t value to 1,2 seconds which corresponds to 50 instances in the first minute. The load on the system will be steadily increased for the subsequent four minutes upon which in the final sixth minute will leave the same load as in the fifth minute and thus create a load

plateau. In total around 430 workflow instances will be launched against the system unevenly spread across six minutes. Finally, we will wait further four minutes of catch-up time to collect the results of lagging instances which might have been slowed down due to overload and/or failure at nodes.

The exactly chosen load on the system is based on previous experimental experiences. On the one hand, this load setting enables us to create critical situations in the environment, such as overload at nodes, for respective configurations. On the other hand, the same load setting is also sufficient to show the effects of the various concepts our work, such as such as scalability, robustness, self-optimization etc., aimed at overcoming those critical situations. The metric that we use to measure and analyze the system execution output for all experiments is throughput.

Workflow instance throughput – denoted as P – measures the number of instances that have been completed each 30 seconds. Moreover, out of the number of completed instances we further count the number of unsuccessfully finished ones, i.e., the ones that have been aborted due to execution problems in the environment. The throughput metric allows us to quantitatively analyze the effects of the self-healing (i.e., SafetyRing) and self-optimization (dynamic service deployment) contributions our work for the control flow.

Finally, we designate the y axis to the workflow instance throughput value P and the x axis to the time dimension for illustration purposes of the upcoming experiments. Whereas each experiment is depicted with a separate line of different color that is labeled according to the node configuration inside the environment.

7.1.2 Baseline Evaluation of the Control Flow

In this subsection we show the default system behavior of the control flow execution. The contributions of our work mainly address drawbacks of distributed workflow management, in terms of control flow execution robustness and performance. This implies that more experiments addressing the issues of the control flow have been conducted than the data flow. The extent of control flow experiments spans two sections of this chapter. In order to make their results comparable we have to extract the default system behavior first. That is why we dedicate the experimental results of baseline system behavior a separate section, that will be used as reference in the subsequent sections so as to quantify the gain of our work.

The goal of this experiment is to show the baseline system behavior of the control flow execution with regard to the output metric P (i.e., workflow instance throughput) when the system is subjected to volatility and heterogeneity. Precisely, in this experiment we will use 11 and 20 evaluation nodes. In separate runs we will configure the environment to contain 25%, 50% and 75% of mobile nodes. For a zero mobile node configuration we will further adjust the environment to contain 0%, 25% and 50% of faulty nodes in separate runs. Hence, our baseline system configuration space will be defined by the by the number of nodes N, its possible proportions of mobile nodes M, its possible proportions of faulty nodes F and this configuration space will be completely covered in our experiments. The complete list of all evaluated experimental system configurations is shown in the following table:

Table 7.2: Baseline evaluation system configurations

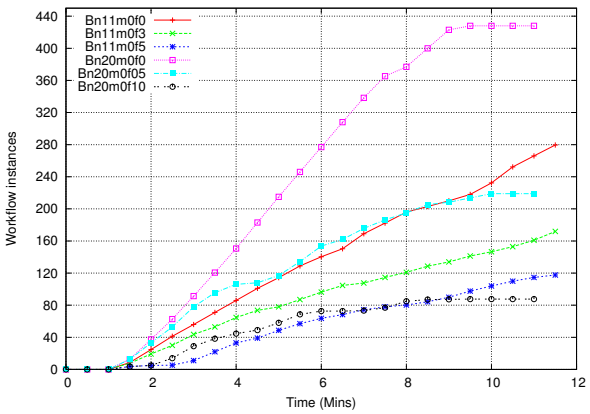
Experiment Id	Number of nodes (N)	Number of mobile nodes (M)	Number of faulty nodes (F)
Bn11m0f0	11	0	0
Bn11m0f3	11	0	3
Bn11m0f5	11	0	5
Bn11m3f0	11	3	0
Bn11m5f0	11	5	0
Bn11m8f0	11	8	0
Bn20m0f0	20	0	0
Bn20m0f5	20	0	5
Bn20m0f10	20	0	10
Bn20m5f0	20	5	0
Bn20m10f0	20	10	0
Bn20m15f0	20	15	0

Expected Results

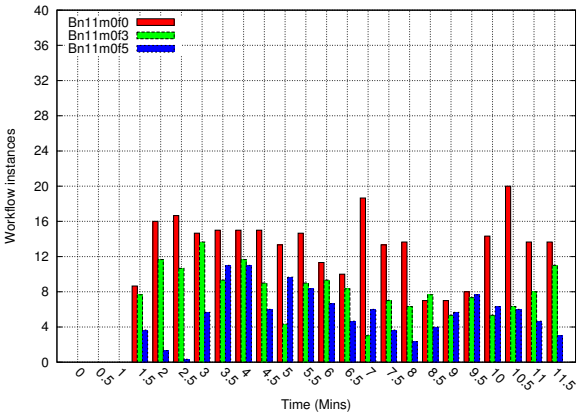
The baseline system should yield results that feature a high system performance, in terms of P. All workflow instances should finish within the extended catch-up time. The throughput performance of the 20 node setting should be slightly better than the 11 node setting due to the scalable nature of the distributed workflow execution model and all workflow instances should finish with even less delay. In case mobile nodes are introduced into the environment, the throughput should slightly suffer as instances that traverse the mobile nodes will lag behind due to the increased latency and should require more time for completion. Of course the throughput degradation should coincide with the increase of mobile nodes in the environment. If faulty nodes are introduced into the environment, the throughput should severely suffer and should not coincide with the number of launched instances. This is due to the fact that workflow instances should disappear together with the faulty node at moment of failure in case they are late-bound for an invocation. Analogously, throughput degradation should coincide with the increase of faulty nodes in the environment.

Experimental Results

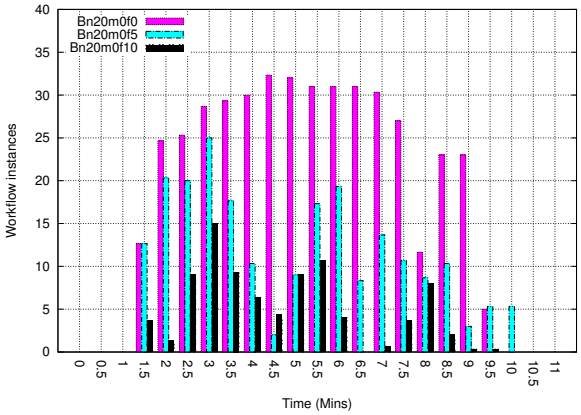
Figure 7.2 depicts the evaluation results of baseline the system performance w.r.t. P when faulty nodes are introduced into the system. Precisely, Figure 7.2(a), shows the number of aggregated workflow instances that have finished per 30 *sec* for all node configurations. Figure 7.2(b), shows the number of finished workflow instances per 30 *sec* for all 11 node configurations. Figure 7.2(c), shows the number of finished workflow instances per 30 *sec* for all 20 node configurations. The results are summarized as follows:



(a) Aggregated workflow instances that have finished



(b) Finished workflow instances per 30 sec for 11 nodes



(c) Finished workflow instances per 30 sec for 20 nodes

Figure 7.2: Workflow instance throughput for with faulty nodes for the baseline

- *11 node setting with failures* – As we can observe from the eleven node setting (i.e., $Bn11m0f0$) that the maximum throughput performance of the system lies steadily at around 15 workflow instances per 30 *sec* (Figure 7.2(b)). Due to the fact that the computationally more intensive services F and G are deployed only once and they act as throughput bottlenecks. This means that system manages to linearly process around 280 workflow instances after 10 minutes of execution time in a failure-free scenario (i.e., $Bn11m0f0$) which does not meet our expectation. Observe from Figure 7.2(b) that after the eight minute the P drops to 8 instances per 30 *sec* due to the stopped influx of new workflow instances into the system. After one minute the throughput rises again to the maximum capacity as delayed workflow instances which are queued at the bottleneck service instances in the system become subject to subsequent processing.

  When failures are introduced into the system (i.e., $Bn11m0f3$ and $Bn11m0f3$) the throughput significantly decreases as expected. Already with the introduction of 25% of faulty nodes (i.e., $Bn11m0f3$) the throughput declines almost by 50% to

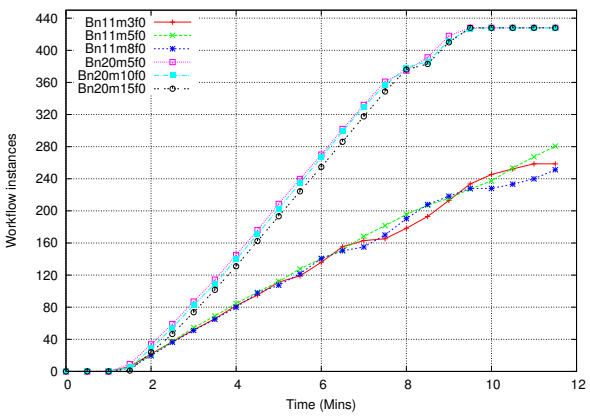
around 15 workflow instances per 30 *sec* (i.e., $\sim$ 170 instances after 10 minutes). The difference in workflow instances among the two experiments reflects in loss of them due to a lack of an effective fault-tolerance mechanism by the baseline system. When even more faulty nodes are added to the system (i.e., $Bn11m0f5$) the throughput in workflow instances even further increases to $\sim$ 120 instances after 10 minutes. The difference in 200 workflow instances as compared to $Bn11m0f0$ is also lost due to failure of nodes.

- *20 node setting* – When the system is scaled up to 20 nodes the system performs as expected with regard to throughput. For the failure-free case (i.e., $Bn20m0f0$) we can observe the benefits of the scalable architecture of OSIRIS. Namely, P doubles to a steady 30 workflow instances per 30 *sec* (Figure 7.2(c)) as compared to the 11 node case. Given the increase in service instance number of type F and G the orchestrator nodes can at late-bing time direct workflow instances towards the newly added service instances and thus provide load balancing among them. As a result all workflow instances can be processed by the system after 2, 5 minutes of catch-up time. The same drop in throughput can be also observed for the 20 node case once the influx of workflow instances stops. The scaled up configuration the system manages to resume maximum capacity faster as workflow instance queues at nodes should be shorter due to higher redundancy of service instance numbers.

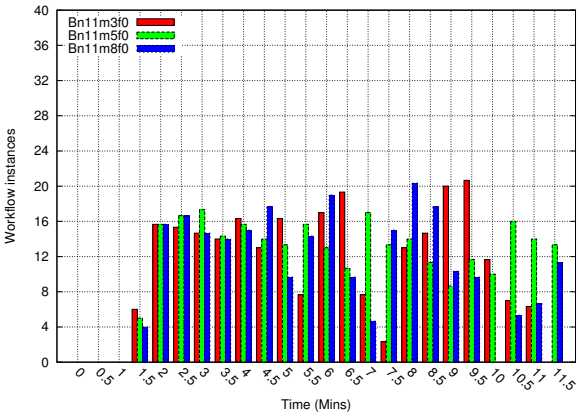
  Naturally, workflow instances are lost when failures are introduced into the system (i.e., $Bn20m0f5$ and $Bn11m0f10$) as well. Moreover, the same proportions of lost workflow instances are observed as well for this configuration. That is, with the introduction of 25% of faulty nodes P drops by 50% to 210 workflow instances (i.e., $Bn20m0f5$), whereas with of 50% of faulty nodes P drops by 50% to 90 workflow instances (i.e., $Bn20m0f10$) as compared to $Bn20m0f5$. The only difference to the 11 node configuration scenarios is that the losses ca be observed more clearly since after some time a throughput plateau forms, i.e., for $Bn20m0f5$ after 10 minutes and for $Bn20m0f10$ after 8 minutes. This implies that no workflow instances are finished any more after these points in time due to failure. The same observations can also been observed in Figure 7.2(c) after the minutes 10 (for $Bn20m0f5$) and 8 (for $Bn20m0f10$).

Figure 7.3 depicts the evaluation results of the baseline system performance with regard to P when mobile nodes are introduced into the system. Precisely, Figure 7.3(a), shows the number of aggregated workflow instances that have finished per 30 *sec* for all node configurations. Figure 7.3(b), shows the number of finished workflow instances per 30 *sec* for all 11 node configurations. Figure 7.3(c), shows the number of finished workflow instances per 30 *sec* for all 20 node configurations. The results are summarized as follows:

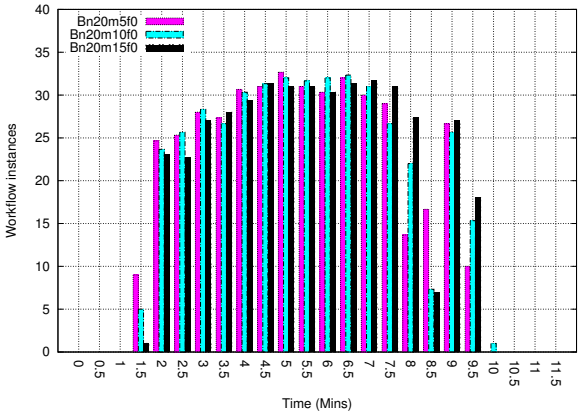
- *11 node setting with mobility* – As we can observe from the eleven node setting (i.e., $Bn11m3f0$) with the smallest percentage of mobile nodes, mobility has an impact on P, albeit unexpectedly a very small one. Precisely the average throughput per 30 *sec* lies also around 15 workflow instances (Figure 7.3(b)) which yields a total P of 250 workflow instances after 10 minutes of execution time. This means



(a) Aggregated workflow instances that have finished



(b) Finished workflow instances per 30 *sec* for 11 nodes



(c) Finished workflow instances per 30 *sec* for 20 nodes

Figure 7.3: Workflow instance throughput for with mobile nodes for the baseline

the increase of M by 25% results in a decrease of P by only 10%. This could be attributed to the fact that the workflow instances we chose do not convey any meaningful data. Hence, the migration of workflow instances of small data sizes does not pose a problem for the distributed execution model of OSIRIS.

Unexpectedly, this is even more obvious when the percentage of mobile nodes is increased to 50%. As we can see the corresponding configuration $Bn11m5f0$ features even a better performance as compared to the $Bn11m3f0$. We believe that the increase in mobile nodes has slowed down the inflow rate of which workflow instances at the bottleneck service instances F and G. As a result, the load on them decreased making it possible to catch-up with the already queued workflow instances.

As expected with the additional increase of mobile nodes to 80% (i.e., $Bn11m8f0$) P decreases again but very slightly as well. That is, $Bn11m8f0$ features almost the same performance as the configuration $Bn11m8f0$. The increase of mobile nodes

should have decreased the transmission of workflow instances sufficient so that P decreases again as compared to $Bn11m5f0$.

In general, the same performance drops can be observed around the seventh minute for the configurations $Bn11m3f0$ and $Bn11m8f0$ as well. Whereas, $Bn11m5f0$ features a rather constant throughput over the whole evaluation interval. This underpins us in our opinion that the service instances F and B have been slightly relieved by the reduced transmission rate of workflow instances – just enough to faster complete the queued ones.

- *20 node setting with mobility* – When the environment is scaled up to 20 nodes the system behaves as expected. P increases to around 30 workflow instances per 30 *sec* (Figure 7.3(c)) so as to complete the execution of all workflow instances after 2.5 minutes of catch-up time. This is true for all mobile node configurations, i.e., $Bn20m8f5$, $Bn20m8f10$ and $Bn20m8f15$. Even the same performance drops can be observed for the configurations, albeit one minute later as compared to the eleven node configurations. Although the increase in mobile nodes for $Bn20m8f10$ to 25% and $Bn20m8f15$ to 80% slightly decreases P this is barely visible on the figure. Just like in the case of the 11 node configuration we attribute this fact to the omission of meaningful data within the workflow instances.

We conclude the system to be generally performing as expected with regard to control flow migration. Simply put, it scales and significantly losses workflow instance in face of node failure. However, mobility of nodes does not pose a challenge to system if no voluminous data is shipped around and performance seems to suffer from bottlenecks for low node configurations if service types are not distributed optimally.

7.1.3 Environment for Streaming Evaluations

To evaluate the continuous data flow w.r.t. its dynamic runtime execution behavior we confine all our experiments to a common workflow definition, a common execution environment and a common experimental procedure. The common experimental characteristics are subject to a detailed discussion in the remainder of this subsection.

Workflow Definition for Streaming Evaluations

The workflow definition we use for the evaluation stems from the dynamic emergency management application scenario of Section 2 and mimics a simple and data processing chain. Figure 7.4 depicts the used workflow definition. As the figure shows the evaluation workflow definition is composed a variable number of activities out of which there are only three distinct service types, i.e., *TestSensor, Intermediate* and *Sink*. Each one of them simulates continuous processing of data streams such that they differ in their semantics for the overall data flow. The semantics of each individual service type are summarized as follows:

- *TestSensor* – This service type corresponds to the source of the continuous data flow. That is, only the start activity can precede it. It continuously produces (i.e.,

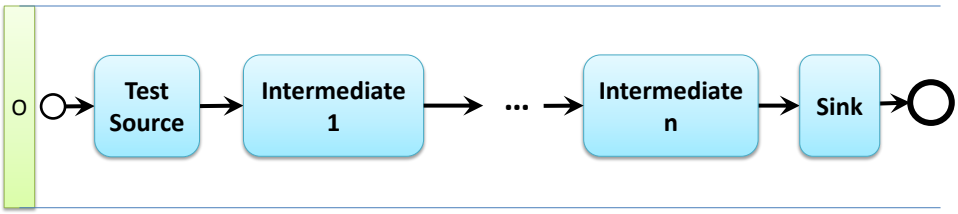


Figure 7.4: The workflow definition in BPMN notation for the evaluation of the data flow.

streams) data items such that the payload of each data item corresponds to its creation timestamp. The TestSensor service creates data items at a rate of 100 data items per second that are forwarded to the output stream.

- *Intermediate* – The TestSensor is followed by an arbitrary number of Intermediate service types. In terms of business logic, this service type performs simple data item forwarding and data stream consistency validation. That is, each data item from the input data stream is simply forwarded to the output data stream. However, the Intermediate service type also features an internal state that corresponds to a counter of the currently processed data item. This counter is exploited for the detection of inconsistencies in the data stream which can occur due to lost data items. To this end the business logic of the service type indicates the next valid data item which has to be processed so as to be consistent.

- *Sink* – The Sink service type concludes the continuous data flow structuring. That is, only an end activity can succeed it. Moreover, this service types consumes all data items and detects inconsistencies in a similar fashion as the Intermediate service type does. The Sink service type additionally validates whether all data items of the data flow have arrived to it as well.

As Figure 7.4 already suggests multiple Intermediate service types can be added to the structure of the data flow. Thereby, each Intermediation service type can only feature one predecessor and one successor as determined by the data flow precedence order. We limit ourselves in our experiments to two and three Intermediate service type instantiations which are evaluated separately.

To achieve a true flow of data among the streaming service instances, we spread the services across all nodes such that no node can stream data items to itself in deploying only one service type per node. To this end we apply the modulo operator on all peers as follows:

$$Peer_{id} \textbf{ mod } |Services|$$

Each distinct outcome of the modulo operator is designated to one service type. We apply the modulo operator starting from peer one. As usual the zeroth peer is super-peer which hosts all necessary global metadata repositories (i.e., the Repository component). In the context of data flow evaluations the supper-peer contains the TestSensor as well. This implies that all service types except for the TestSensor are redundantly deployed in the environment. This is along the lines of the passive-standby recovery mechanism which exploits service instance redundancy for safe of data flow reliability. Depending

on the number of different Intermediate service types the level of redundancy is going to vary. Since we apply either two or three distinct Intermediate service types in separate configurations our experiments will feature two different deployments of services. These are summarized by the Table 7.4 and Table 7.3

Table 7.3: Services distribution at nodes for two distinct Intermediate service types

Service	Peers
$Super - peer$	Peer 0
$TestSensor$	Peer 0
$Intermediate_1$	Peer 1, Peer 4, Peer 7, Peer 10, Peer 13, Peer 16, Peer 19
$Intermediate_2$	Peer 2, Peer 5, Peer 8, Peer 11, Peer 14, Peer 17
$Sink$	Peer 3, Peer 6, Peer 9, Peer 12, Peer 15, Peer 18

Table 7.4: Services distribution at nodes for three distinct Intermediate service types

Service	Peers
$Super - peer$	Peer 0
$TestSensor$	Peer 0
$Intermediate_1$	Peer 1, Peer 5, Peer 9, Peer 13, Peer 17
$Intermediate_2$	Peer 2, Peer 6, Peer 10, Peer 14, Peer 18
$Intermediate_3$	Peer 3, Peer 7, Peer 11, Peer 15, Peer 19
$Sink$	Peer 4, Peer 8, Peer 12, Peer 16

Note, that all nodes also feature the orchestration service and all the corresponding *OSIRIS-rSE* components.

Execution Environment

The execution environment we have chosen to run our data flow experiments against is fairly simple. It is comprised of 20 equally equipped nodes. Each node is an Amazon EC2 virtual device instance. The instance type used for the data flow experiments is t1.micro[2]. Note, that this instance type significantly differs from the c1.medium which has been used for the control flow experiments. A t1.micro node features only very limited CPU, memory and bandwidth resources such that they are well suited as mobile sensor devices. Simply put, a t1.micro node features:

- One virtual CPU of 1.0-1.2 GHz performance,
- Main memory of 615 MB,

[2]http://aws.amazon.com/ec2/previous-generation/

- Low permanent storage of 9.0 GB
- Very low network bandwidth performance of $\sim 20Mbs$.

Since we are evaluating only the self-healing features of our work only uniformity of redundant nodes, in term of hardware resource, guarantees elicitation of the conceptual effects.

Volatility inside the computational environment, such as in the case of the control flow evaluations, is introduced again by means of faulty nodes. Recap, faulty nodes are simulated by stopping the OSIRIS processes at the normal nodes at a random point in time. The precise stopping moment is determined by the same uniformly distributed random function that remains within the limits of $0 - 320$ seconds. The faulty nodes eventually rejoin the execution environment after 60 seconds of breakdown time and are again subject to a new random stop. Volatility of the execution environment is achieved by configuring in separate evaluation runs a certain amount of normal nodes to be faulty. Precisely, all service instances of Intermediate type are configured to be faulty.

Experimental Procedure

The simple composition of the evaluation workflows is designed to correspond to a simple data transfer request by some system end-user that lasts over a small period of time. In the context of the emergency management scenario the simple workflow corresponds to the scenario in which sensor data produced at firemen is streamed to the trucks for aggregation purposes. The data stream is thus mimicked by creating a workflow instance which activates all streaming service instances in conjunction with their backup nodes in the course of the control flow. Upon activation of the streaming instances the continuous data flow is initiated and it lasts for six minutes. To this end the super-peer initiates the workflow instance and serves as the streaming data source by solely hosting the TestService service type.

To elicit the self-healing features of the system we will enable the faulty nodes once the continuous data flow has been initiated. From this point forward failures of random nodes and at unforeseeable points in time can happen. That is, only the service instances of type TestSensor and Sink are exempt from the failures. As elaborated before the data source is assumed to be fails-safe and thus it cannot failed. On the other hand, a service instance of Sink type serves as a measurement and consistency validation node by counting the incoming data items and assessing them for their order. Hence, it cannot be failed as well.

Data item throughput – denoted as D – measures the number of instances that have been completed each 30 seconds at the Sink service instance. The data throughput metric allows us to quantitatively analyze the effects of the self-healing (i.e., redundant passive standby) contributions our work w.r.t. the data flow.

In separate runs we evaluate D for different workflow settings of encompassed Intermediate service types. In case we use two Intermediate service instances the workflow instance features four active service instances in total at the same time. In case we use three intermediate service instances the total number of streaming service instances sums up to five.

By increasing the number of Intermediate service types we directly expose the data flow to more volatility. From the tables 7.3 and 7.4 we can conclude that in the case of 5 active service instances more nodes are exposed to random failure. Recap, all service instances of type Intermediate are set to be faulty, independent of their activity status, at all times. This implies that for Table 7.4 16 Intermediate service instances are affected by volatility whereas for Table 7.3 13 service instances are affected.

For each of the workflow structure cases we further vary the number of backup nodes for each active services from one to three so as to address the volatility with the redundancy. The node configuration featuring just one node corresponds to the baseline system output and is used as a reference for our work. Each backup node configuration is validated by means of a separate experiment.

This implies that out of the 16 failure affected service instances (in Table 7.4) 9 nodes are directly participating to the data flow (either as active service instances or backups) for the 2 backup node case and 12 are directly participating to the data flow for the 3 backup node case. On the other hand, out of the 14 failure affected service instances (in Table 7.3) 6 nodes are directly participating to the data flow for the 2 backup node case and 8 are directly participating to the data flow for the 3 backup node case. Based on this numbers we conclude that by increasing the number of active service instances the data flow is more likely to be affected by node failure.

Finally, we designate the y axis to the data throughput value D and the x axis to the time dimension for illustration purposes of the upcoming experiments. Whereas each experiment is depicted with a separate line of different color that is labeled according to the node configuration inside the environment.

7.2 Evaluation of the Self-healing Execution

In this section we evaluate the qualitative gain of our work with regard to reliability of the control flow and the continuous data flow via a series of experiments. Thereby, we exploit the execution environment, in terms common configuration characteristics, of the baseline system so as to apply our concepts on top of it. Since we provide two separate solutions for the reliability of the control flow (i.e., the Safety-Ring) and the reliability of the steaming data (i.e., redundant passive-standby) we separate our experiments with that regard.

7.2.1 Safety-Ring Evaluations

The control flow is supported by means of the Safety-Ring so as to offer reliability. Hence, we provide in this section quantitative results of the reliability that Safety-Ring offers in a set of experiments. In case mobile nodes are added to the environment we enable the Compass feature of Safety-Ring. The resulting effects of Compass are provided by means of a separate experiments set.

In this experiment set we will evaluate the system behavior with regard to the execution output metric P, with the enabled Safety-Ring as compared to the baseline. Therefore, we will use the same experimental system configurations, however extended with

another configuration parameter, i.e., the number of SR-nodes. In separate evaluation runs we will configure the system to additionally contain 50% and 100% of SR-nodes out of all nodes N. Mobile nodes are omitted from this experiments set. Hence, the complete list of all evaluated experimental system configurations is shown in the following table:

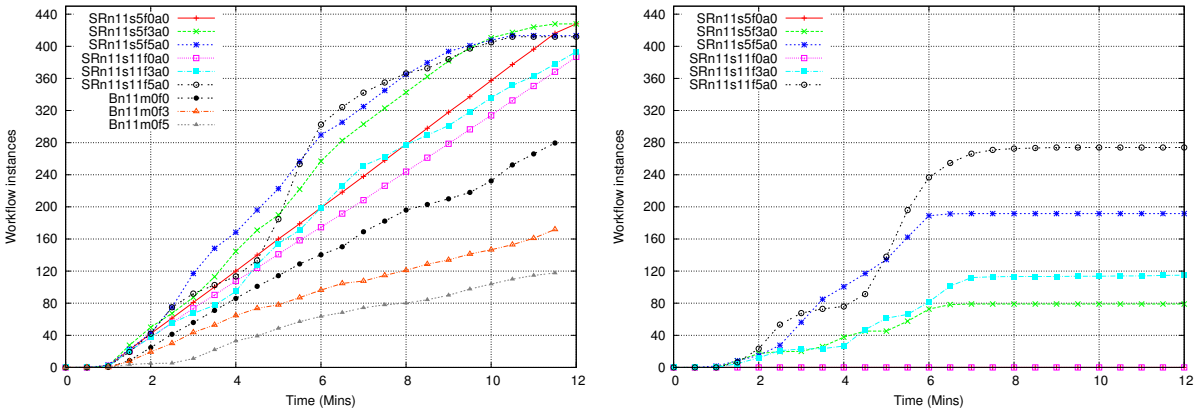
Table 7.5: Safety-Ring enabled evaluation system configurations

Experiment Id	Number of nodes (N)	Number of Safety-Ring nodes (S)	Number of faulty nodes (F)	Safety-Ring self-configuring (A)
SRn11s5f0a0	11	5	0	no
SRn11s5f3a0	11	5	3	no
SRn11s5f5a0	11	5	5	no
SRn11s11f0a0	11	11	0	no
SRn11s11f3a0	11	11	3	no
SRn11s11f5a0	11	11	5	no
SRn20s10f0a0	20	10	0	no
SRn20s10f5a0	20	10	5	no
SRn20s10f10a0	20	10	10	no
SRn20s20f0a0	20	20	0	no
SRn20s20f5a0	20	20	5	no
SRn20s20f10a0	20	20	10	no

Since Safety-Ring necessitates a certain degree of workflow instance metadata redundancy so as to guarantee availability in the presence of failure, we select a replication factor of $r = 3$. This mean that metadata of all workflow instances is replicated on three different SR-nodes. A higher replication factor guarantees improved robustness to failures, however it is only applicable environments of higher node numbers. For instance, a higher replication factor of e.g., 5 would necessitate replications to the majority of nodes inside the environment for the low Safety-Ring node configurations.

Expected Results

The expectation of our experiments towards the Safety-Ring enabled baseline system is that more workflow instances will be saved from node failure. The throughput performance of the system should be lower due to the overhead of the Safety-Ring. That, is P should be visibly lower (delayed) as compared to the baseline. This decline in P should however improve with the upscaling of SR-nodes. Even at 11 nodes the increase in SR-nodes should yield a better P as load can be shared among them. With the upscaling of the Safety-Ring node to 20 nodes P is expected to be very similar to the baseline performance.



(a) Aggregated workflow instances that have finished

(b) Aggregated workflow instances that have been aborted

Figure 7.5: Workflow instance throughput for 11 node configurations with faulty nodes for the baseline and Safety-Ring

In case faulty nodes are introduced into the system, throughput should be visibly improved due to the Safety-Ring fault recovery. Out of all finished instances a significant proportion should end with an unsuccessful result (i.e., with an abort) for the 11 node configuration as the environment does not provide substitutions for recovery. Of course with the upscaling of the system recovery efficiency of Safety-Ring should be much better.

As compared to the baseline, the system should only exhibit a minimal number of lost workflow instances due to failure. Only when the system suffers from a simultaneous failure of the majority of SR-nodes, belonging to the same equivalence class, workflow instances can be lost. This is due to the fact that Safety-Ring exploits Paxos commit for the replication of data items, which in turn guarantees reliability of F given $2F + 1$ SR-nodes. In our case this implies that only one SR-node of a equivalence class is allowed to fail so as to guarantee reliability. Of course the throughput degradation, workflow instance abortion rates and disappearance numbers should coincide with the increase of faulty nodes in the environment.

Experimental Results

Figure 7.5 depicts the evaluation results of Safety-Ring enabled system performance with regard to P of 11 nodes as compared to the baseline. Precisely, Figure 7.5(a), shows the number of totally finished workflow instances per 30 *sec*. Figure 7.5(b), shows the number of totally aborted workflow instances per 30 *sec*.

- *5 SR-node setting with failures* – As we can observe from the Figure 7.5 some unexpected results are returned when the Safety-Ring is put into play. Firstly, when only five SR-nodes are activated (i.e., $SRn11s5f0a0$) P increases to about steady 20 workflow instances per 30 *sec*, which results in a completion of all workflow instances in eleven minutes. This is somehow surprising as the expectation was

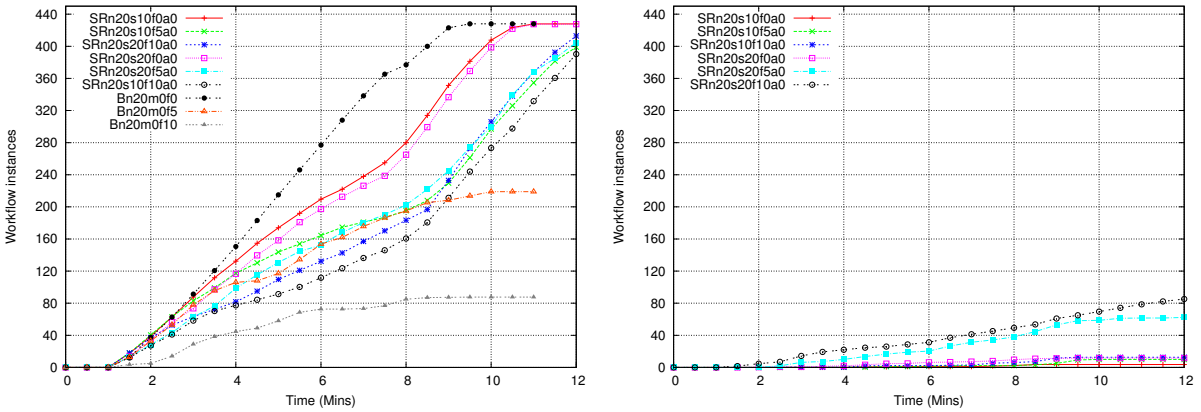
for the Safety-Ring to lower the throughput as compared to $Bn11m0f0$. The explanation lies in the fact that Safety-Ring completely forces through the Peer-to-Peer execution style and does query centralized metadata repositories at runtime. Recap, the baseline has to query the join metadata from the repository at runtime so as to obtain the join service instance information. In Safety-Ring late-binding of the join service instances is carried out in a distributed fashion by the SR-nodes which clearly boosts throughput.

When failures are introduced into the system (i.e., $SRn11s5f3a0$) the system finally behaves as expected. As compared to the baseline (i.e., $Bn11m0f3$) almost all workflow instances can be recovered. Thereby, the system manages to complete all workflow instances after 11.5 minutes, We can also observe from Figure 7.5(b) that out of the completed workflow instances a certain amount of them had to be aborted. For $SRn11s5f3a0$ around 80. This is due to the fact that at recovery time no substitution service instances were present so the workflow instances could not be late-bound and thus had to be aborted.

When even more failures are introduced into the system (i.e., $SRn11s5f5a0$) behaves expectedly, as well. The throughput is the about the same 410 workflow instances after ten minutes. This high throughput can be explained by studying Figure 7.5(b) which feature far more aborted workflow instances than in the configurations $SRn11s5f3a0$. Given more faulty service instances more workflow instances have to be aborted at runtime as well as there are less substitution late-binding possibilities.

- *11 SR-node setting with failures* The same system trends can be observed when the Safety-Ring is extended to all nodes (i.e., $SRn11s11f0a0$), even there the throughput is significantly better than the baseline for the same reason of omitted join repository. However, as compared to $SRn11s5f0a0$, P of $SRn11s11f0a0$ slightly decreases to about steady 18 workflow instances per 30 *sec*, i.e., 380 workflow instances after eleven minutes. We explain this trend with the extension of the Safety-Ring over to the service instances F and G. These are as already mentioned more computationally intensive and thus also consume significant local resources. Although system services such as Safety-Ring are more privileged over the application services, in terms of thread priority, the excessive computational requirements of F and G make themselves felt when in concurrence to other services.

  When failures are introduced into the system the full SR-node configuration (i.e., $SRn11s5f3a0$) manages to complete $\sim$ 390 instances after 12 minutes. The difference in P among the two configurations can be explained by the fact that in $SRn11s11f3a0$ also three SR-nodes failed which had to be recovered themselves before workflow instances could be, thus the delay in throughput. The configuration $SRn11s11f3a0$ features more aborted workflow instances due to recovery of failed SR-nodes which implies reconfiguration of the Safety-Ring. In the meantime update of workflow instance state (i.e., Paxos commit writes) at SR-nodes affected by the reconfiguration is temporarily not possible (e.g., compass table inconsistencies, missing data items at the SR-node) and thus have to be aborted after a finite number of times.



(a) Aggregated workflow instances that have finished

(b) Aggregated workflow instances that have been aborted

Figure 7.6: Workflow instance throughput for 20 node configurations with faulty nodes for the baseline and Safety-Ring

In case of the full SR-node configuration (i.e., $SRn11s11f5a0$) is subjected to even more failure the throughput is the about the same (i.e., 410 workflow instances after ten minutes) as $SRn11s5f5a0$. However, more failed SR-nodes inflict even more aborted workflow instances (i.e., 80 more) due to the even bigger numbers of affected SR-nodes with the Safety-Ring reconfiguration overhead at runtime. As a consequence, even more transactions will not be possible and thus will have to be aborted.

Figure 7.6 depicts the evaluation results of Safety-Ring enabled system performance with regard to P of 20 nodes as compared to the baseline. Precisely, Figure 7.6(a), shows the number of totally finished workflow instances per 30 *sec*. Figure 7.6(b), shows the number of totally aborted workflow instances per 30 *sec*. The results are summarized as follows:

- *10 SR-node setting with failures* – Upscaling the system (i.e., $SRn20s10f0a0$) brings about the same workflow instance throughput behavior of the Safety-Ring. The performance of the system slightly improves by one minute (i.e., in total ten minutes) to complete the execution of all workflow instances as compared to the 11 node Safety-Ring setting. Thereby, the same drop in throughput (to on average 12 per 30 *sec*) due to stopped influx of workflow instances can be observed around the sixth minute as well. However, in this setting the system behaves more as expected when compared to the baseline. Precisely, P is notably lower (i.e., $\sim$ 20 workflow instances per 30 *sec*) than the baseline (i.e., $Bn20m0f0$ – $\sim$ 30 workflow instances per 30 *sec*) due to the overhead of the Safety-Ring. The fact that $Bn20m0f0$ significantly outperforms $Bn10m0f0$ leads us to believe that in general the combination of bottleneck service instances and bottleneck repositories is bad for throughput whereas just one bottleneck is not given a scalable architecture. Safety-Ring features no bottleneck entities.

In face of 25% of faulty nodes (i.e., $SRn2010f5a0$) the system behaves much better, in terms of recovery. With the presence of more substitution service instances the Safety-Ring can recover almost all workflow instances (i.e., 400) after eleven minutes. Out of the recovered ones only $\sim$ 10 had to be aborted whereas the other ones could be successfully late-bound to substitution service instances. We attribute the aborted instances to metadata propagation either delays or transactional conflicts. The former implies that due to metadata on substitution service instances was not propagated on time to the SR-nodes such that they concluded no substitution service to be available. The latter implies, that parallel activities issued transactional updates at the same time thus causing repeated conflicts which had to be aborted after a finite number of times.

The increase of the percentage of faulty nodes to 50% ($SRn2010f10a0$) brings about the same performance ,i.e., 410 workflow instances after eleven minutes. Figure 7.6(b) shows $SRn2010f10a0$ feature a lower P during workflow instance influx time, due higher recovery overhead, but manages to catch-up once the influx is stopped (i.e., after the seventh minute).

- *20 SR-node setting with failures* The same trend can be observed with increase of SR-node numbers, i.e., $SRn20s10f0a0$. The throughput is about the same as compared with $SRn20s10f0a0$ (i.e., all workflow instances are completed ten minutes of execution time), although it is slightly lower during the whole course of the execution. The reason is the same as in the 11 node setting – more SR-nodes are collocated with the computationally intensive service instances F and G.

  Introduction of faulty nodes (i.e., $SRn2020f5a0$ and $SRn2020f10a0$) brings about the same trends all over again. A majority of workflow instances (i.e., 400 for $SRn2020f5a0$ and 390 for $SRn2020f10a0$) is recovered after eleven minutes of execution time due to the availability of substitution service instances. As in the case of $SRn2010f10a0$ the failure of SR-node entails their recovery first before workflow instances can be recovered. As expected for these two settings the abortion number is visibly higher – 60 for $SRn2020f5a0$ and 85 for $SRn2020f10a0$. Naturally, more failed SR-nodes imply higher Safety-Ring recovery overhead, during which transactions are more likely to be aborted along with the featuring workflow instances.

Based on the shown experiments we conclude the Safety-Ring to be performing as expected. Simply put, the Safety-Ring provides reliability of the control flow. Even for small node configurations reliability can be achieved, albeit at the cost of frequent aborts. Whereas in upscaled configurations the Safety-Ring behaves just as it should. It recovers the majority of workflow instances at the price of notably reduced throughput. However, the scalable architecture of it allows for the improvement of throughput with the increase of SR-node numbers and for small configurations even better performance than the baseline.

7.2.2 Compass Evaluations

In this experiment we will experiment with a Safety-Ring enabled system when mobile nodes are introduced with regard to the execution output metric P. In order to observe node latency effects on the execution, we will subject all nodes to the Safety-Ring at all times and omit the faulty ones. In separate evaluation runs for different mobility configurations we will enable the Compass extension so as to compare the mobility enhanced Safety-Ring with the baseline system. Hence, the experimental system configuration set for this experiment is shown in the following table:

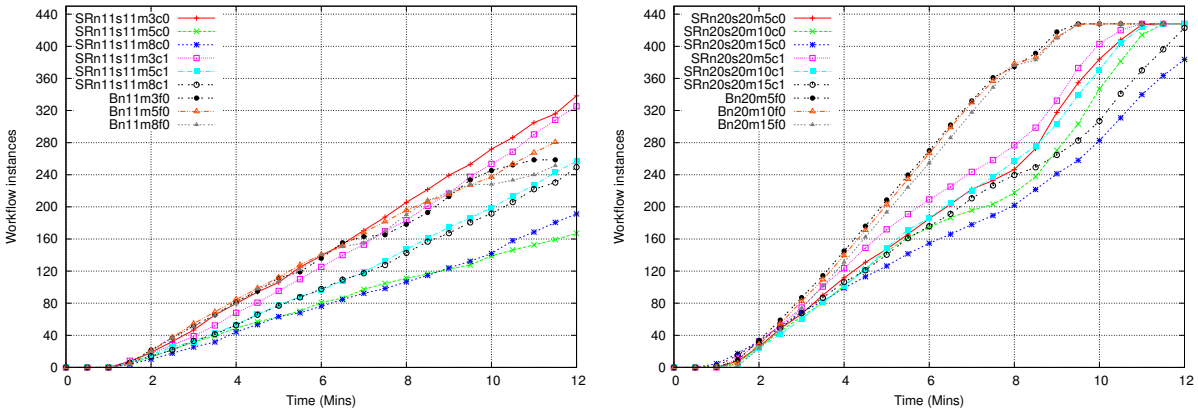
Table 7.6: Baseline evaluation system configurations

Experiment Id	Number of nodes (N)	Number of Safety-Ring nodes (S)	Number of mobile nodes (M)	Compass (C)
SRn11s11m3c0	11	11	3	no
SRn11s11m5c0	11	11	5	no
SRn11s11m8c0	11	11	8	no
SRn11s11m3c1	11	11	3	yes
SRn11s11m5c1	11	11	5	yes
SRn11s11m8c1	11	11	8	yes
SRn20s20m5c0	20	20	5	no
SRn20s20m10c0	20	20	10	no
SRn20s20m15c0	20	20	15	no
SRn20s20m5c1	20	20	5	yes
SRn20s20m10c1	20	20	10	yes
SRn20s20m15c1	20	20	15	yes

Note, that the Compass Table compression algorithm is not applied in the experiments.

Expected Results

The expectation of our experiments towards the Compass is that the workflow instances throughput will be improved as compared to plain Safety-Ring when subjected to mobility. First of all we expect the system to significantly decrease in performance when plain Safety-Ring is subjected to mobile nodes. The combination of mobile node traversal in conjunction with the overhead of Safety-Ring should heavily slow down the forwarding of workflow instances. The Safety-Ring maintenance and enforcement algorithms are rather message and data intensive and should claim significant bandwidth resources. Thereby, the more mobile nodes we add into the system the higher the decrease of P should be. However, the application of Compass should significantly improve the performance of the system, but only compared to the plain Safety-Ring case. When it comes to upscaling of the system a slightly better performance of the system is



(a) Aggregated workflow instances for 11 node configurations that have finished

(b) Aggregated workflow instances for 20 node configurations that have finished

Figure 7.7: Workflow instance throughput for mobile configurations with Safety-Ring and Compass

expected as the Safety-Ring overhead should be distributed among the SR-nodes. Compass on top of an upscaled system should also yield an improved performance as more possibilities for latency optimal path will present themselves.

Experimental Results

Figure 7.7 depicts the evaluation results of Compass enabled system performance with regard to P for all nodes as compared to the plain Safety-Ring baseline. Precisely, Figure 7.7(a), shows the number of totally finished workflow instances per 30 *sec* for 11 SR-nodes. Figure 7.7(b), shows the number of totally aborted workflow instances per 30 *sec* for 20 SR-nodes. The results are summarized as follows:

- *11 SR-node setting with Compass* – As we can observe from the Figure 7.7(a) expected results are returned when the Safety-Ring is subjected to mobility. Even the addition of just three mobile nodes (i.e., $SRn11s11f0m3c0$) decreases the throughput performance of the system significantly. After ten minutes of execution the system features a performance of $P \simeq 320$ which is lower by around 50 workflow instances as compared to a static only system. This fully meets our expectations due to the message intensive interaction with the Safety-Ring in the course of the control flow. Precisely, given a replication factor of three (i.e., $r = 3$) around 30 (as discussed in Section 3.3.2) messages have to be exchanged among at most 6 nodes for a workflow instance to be successfully updated, thus the throughput has to suffer from the reduced bandwidth of nodes. Somehow unexpectedly the activation of Compass (i.e., $SRn11s11f0m3c1$) does not manage to improve throughput for the same configuration. We attribute this to the fact that with only three mobile nodes in the environment no meaningful optimal paths could be found just yet. However, even for this low mobile combinations the activation of the Safety-Ring pays off, in terms of P, as compared to the baseline ($Bn11m3f0$) due to the omission of the centralized join repository.

When mobility in the configuration is increased to five nodes (i.e., $SRn11s11f0m5c0$, $SRn11s11f0m5c1$ and $Bn11m5f0$), the system again behaves as expected. The increased mobility in the system drastically decreases P when the Safety-Ring is enabled to a sheer 160 workflow instances after ten minutes of execution. Note, that the baseline throughput performance (i.e., $Bn11m5f0$) is significantly higher, i.e., $P = 280$ after the same time span. With the increased mobility factor to almost 50% almost every Paxos commit transaction must have been affected by at least one mobile node and thus delayed the execution time. In this configuration however the benefits of Compass are fully shown. With the activation of Compass (i.e., $Bn11m5f0$) the performance of the Safety-Ring is linearly improved to $P = 240$ after ten minutes of execution time. With more mobile nodes more optimal path combinations exist which can be discovered by Compass and consequently exploited at runtime.

The same trends can be observed when the mobility factor is increased to eight nodes – $SRn11s11f0m8c0$, $SRn11s11f0m8c1$ and $Bn11m8f0$. The baseline performance (i.e., $Bn11m8f0$) remains about the same, albeit slightly decreased to $P = 250$ after ten minutes of execution time. The plain Safety-Ring system performance (i.e., $SRn11s11f0m8c0$) is similarly low at $P = 180$. Whereas Compass activation (i.e., $SRn11s11f0m8c1$) boosts workflow instance throughput to $P = 230$ for the same evaluation timespan. Given the even higher mobility in system less static mobile nodes are available thus less optimal paths can be discovered by Compass as compared to the five node configuration, i.e., $SRn11s11f0m5c1$.

- *20 SR-node setting with Compass* – As shown in Figure 7.7(b) upscaling the system to 20 nodes brings about the same performance trends. All baseline mobility configurations (i.e., $Bn20m5f0$, $Bn20m10f0$, $Bn20m15f0$) feature similar performances with the increase of mobility as discussed in Section 7.1.2. Due to its scalable architecture the Safety-Ring is not affected by mobile nodes in the lower mobility cases (i.e., $SRn20s20m5c0$) as it manages to complete all workflow instances within the designated time of ten minutes. Unexpectedly, the activation of Compass (i.e., $SRn20s20m5c1$) for even such a small mobility proportion improves the throughput notably to a $\sim$ 25 workflow instances per 30 *sec*. This leads us to believe that latency optimal data lookup is beneficial for the Safety-Ring even in predominantly static configuration provided that the system is under heavy load. Namely, since overloaded SR-nodes are less responsive at runtime they might be avoided from latency optimal paths by Compass. This why Compass facilitates load-balancing inside the Safety-Ring even if mobile nodes are not predominantly included.

  Only when the amount of mobile nodes is increased to $m = 50\%$ the performance of the plain Safety-Ring (i.e., $SRn20s20m10c0$) notably drops for a 20 node setting. As the figure shows it takes another 30 *sec* for the plain Safety-Ring to complete all workflow instances as compared to the designated evaluation time of ten minutes. Thereby, the system features a throughput which is about $\sim$ 15 workflow instances per 30 *sec* during the whole workflow instance influx time. Only during the catch-up time the system manages to feature a higher throughput which

is still lower as compared to $SRn20s20m10c0$. On the other hand, the activation of Compass (i.e., $SRn20s20m10c1$) improves again the performance to steady 20 workflow instances per 30 *sec* which brings it to about the same performance of the static-only configuration. Almost all instances are completed after ten minutes of execution time.

Increased mobility results in a even bigger decrease of the plain Safety-Ring (i.e., $SRn20s20m15c0$) as expected. Not even after eleven minutes of execution time all workflow instances are completed, resulting in a performance of $P \simeq 380$. The Compass activation helps to mitigate this decrease by providing a rather stable throughput performance which yields an almost complete execution after eleven minutes of execution time.

Based on the shown experiments we conclude the Compass extension of the Safety-Ring to be performing as expected. Simply put, Compass improves control flow throughput performance for mobile environments for almost all node configurations. Thus it helps in mitigating the effects of mobility on the Safety-Ring. It even facilitates load-balancing in upscaled and predominantly static configurations which are under heavy load as well. However, Compass does not manage to improve the performance of the Safety-Ring beyond the mobility induced decrease so to reach baseline performance levels.

7.2.3 Streaming Reliability Evaluations

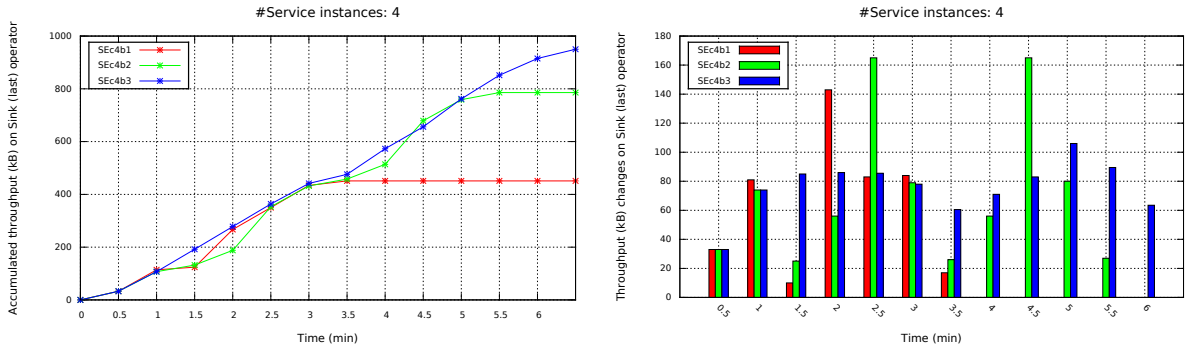
In this experiment set we will evaluate the system behavior with regard to the execution output metric D, with the enabled redundant passive standby as compared to traditional one. Precisely, we will deploy 20 $t1.micro$ evaluation nodes. In separate runs we will configure the workflow definitions to contain either 2 or 3 Intermediate service types. The latter case implies more volatility on the data flow. For each these Intermediate service type case we will configure the active service instances to contain 1, 2 or 3 backup nodes in separate runs. The 1 backup node configuration will correspond to our baseline. Hence, our evaluation configuration space will be defined by the by the number of active service instances O, and its possible backup nodes B. This configuration space will be completely covered in our experiments. The complete list of all evaluated experimental system configurations is shown in Table 7.7.

Expected Results

The system should yield results that feature a high system performance, in terms of D, for the 4 simultaneously active service instances. This is in particular true if no failures are present in the system. When they do occur the baseline setting (i.e., traditional passive standby) is likely to suffer from lower throughput which will eventually result in a complete stop of the data flow. With just one backup node, in the baseline system, random failure of nodes is more likely to simultaneously include an active service instance and its backup node. In such situations the data flow should be interrupted before the system can recover the failures. By increasing the redundancy (i.e., the number of backup nodes B) the system should be more resilient to random failures. The

Table 7.7: Data flow evaluation system configurations

Experiment Id	Number of active service instances (O)	Number of backup nodes (B)
SEo4b1	4	1
SEo4b2	4	2
SEo4b3	4	3
SEo5b1	5	1
SEo5b2	5	2
SEo5b2	5	3



(a) Aggregated data flow for 4 active streaming instances

(b) Data flow throughput for 4 active streaming instances

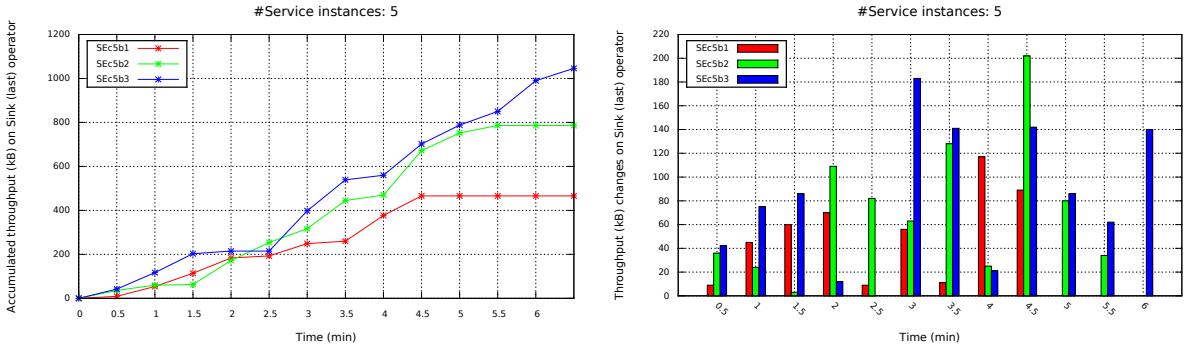
Figure 7.8: Workflow instance throughput for mobile configurations with Safety-Ring and Compass

throughput improvement should coincide with the increase of backup nodes (i.e., B) in the environment.

The system performance, in terms failure robustness, should feature the same trends for the setting with 5 simultaneously activated service instances. Even though the data flow is more likely to be affected by volatility the same trends should emerge from the experiments – the more backup nodes the system has the more resistant it should be to failure. However, the system should in general feature a slightly delayed throughput as well. The data items originating at the source will be additionally delayed by the added third service instance of Intermediate type.

Experimental Results

Figure 7.8 depicts the evaluation results of the system performance with regard to D for 4 simultaneously active service instances. Thereby, Figure 7.8 compares the baseline with the redundant passive-standby enabled system. Precisely, Figure 7.8(a), shows the



(a) Aggregated data flow for 5 active streaming instances

(b) Data flow throughput for 5 active streaming instances

Figure 7.9: Workflow instance throughput for mobile configurations with Safety-Ring and Compass

totally aggregated data throughput D per 30 *sec*. Figure 7.8(b), shows the aggregated data throughput D per 30 *sec*.

Figure 7.9 depicts the evaluation results of the system performance w.r.t. D for 5 simultaneously active service instances. Again, the baseline is compared against a redundant passive-standby enabled system. Precisely, Figure 7.9(a), shows the totally aggregated data throughput D per 30 *sec*. Figure 7.9(b), shows the aggregated data throughput D per 30 *sec*. The results are summarized as follows:

- *4 active service instances* – As the Figure 7.8 shows the system behaves in general as expected when redundant checkpointing of passive-standby is put into play. Even the baseline system (i.e., $SEc4b1$) behaves as expected. Caused by node failures, already after one and a half minutes of execution time the throughput significantly suffers and reaches almost $D \simeq 0$ per 30 *sec*. The baseline system can recover from these failures and catchup the data transmission of the stream in the course of the next minute. That is, the throughput is increased to $D \simeq 140kb$ per 30 *sec* in the third minute of execution time. However, in the third minute another series of simultaneous nodes failures occurs instances which effectively breaks down the data flow for good. After the fourth minute of execution time no more data items are registered at the sink any more for the baseline system. Thereby, the received data volume amounts to $D \simeq 450kb$ after six and a half minutes of execution time.

  Applying the redundant checkpointing to two backup nodes (i.e., $SEc4b2$) helps to improve the reliability of the data flow. The total amount of received data increases as compared to the baseline to almost $D \simeq 800kb$ after six and a half minutes of execution time. The system can in this setting (i.e., $SEc4b2$) even recover from more node failure situations as observed in the second and fourth minute of execution time. Namely, at these two intervals the throughput significantly decreases to $D \simeq 25kb$ per 30 *sec* which implies failure situations that are recovered eventually. The recovery of the system can be best observed in the third and fifth minute during which the throughput substantially increases to $D \simeq 25kb$ per 30 *sec*. This implies that along the lines of the passive-standby recovery the backup

nodes receive next to the regular data items also the unacknowledged ones which is results in the increased throughput. However, even for this setting the volatility in the system, reflected in the faulty node numbers, is too high so as guarantee uninterrupted flow of continuous data. As this figure shows the $SEc4b2$ system configuration also breaks down permanently, albeit only in the fifth minute of execution time.

That is why the checkpoint redundancy is increased to three backup nodes (i.e., $SEc4b3$) in the third experiment. This yields again a better throughput performance of the system, i.e., $D \simeq 950kb$ after six and a half minutes of execution time. Thereby, the system features a steady throughput performance of $D \simeq 80kb$ per 30 *sec* for a long stretch of time that is eventually stopped in the third minute when failure situations occur. These are however successfully recovered while featuring a low drop in throughput performance to $D \simeq 60kb$ per 30 *sec*.

- *5 active service instances* – The same trends can be observed in Figure 7.9 when the chain of streaming service instances is increased by one. As expected the baseline system (i.e., $SEc5b1$) eventually fails after having recovered from a series of failures. In the second (i.e., $D \simeq 10kb$ per 30 *sec*) and third (i.e., $D \simeq 10kb$ per 30 *sec*) minute the baseline system is affected by failures from which it can recover. However, in the fifth minute the baseline system is damaged beyond recovery ba a series of simultaneous failures. The longer execution time as compared to $SEc4b1$ we attribute to the introduced delay by the additional third Intermediate service type. In total the baseline system managed to transmit the about same amount of data (i.e., $D \simeq 450kb$ after six and a half minutes of execution time.) as compared to $SEc4b1$.

  Increase of checkpoint redundancy (i.e., $SEc5b2$) brings about the same results. The system can sustain more node failures, transmit more data but eventually fails. In the experiment of $SEc5b2$ the system recovers from node failures in the first (i.e., $D \simeq 20kb$ per 30 *sec*), second (i.e., $D \simeq 5kb$ per 30 *sec*), third (i.e., $D \simeq 60kb$ per 30 *sec*) and fourth (i.e., $D \simeq 20kb$ per 30 *sec*) minutes but fails in the fifth minute. Thereby, the system managed to transmit about $D \simeq 800kb$ of data after six and a half minutes of execution time. This is somehow surprising, as we expected the additional Intermediate service instance to introduce a throughput delay. We attributed this unexpected throughput improvement to increased checkpoint frequency by the streaming instances which as consequence reduces the amount of data items to be resent (in the output buffers) and thus accelerates the recovery. If more streaming instances exist in the continuous data flow, more checkpoints will be triggered individually but also due to the coordinated checkpointing (i.e., ECOC Algorithm of OSIRIS-SE) by the upstream service instances. Hence, the state at the backup nodes will be freshener, when the failures occur and thus can faster resume the processing of delayed data items.

  Even further increased redundancy ($SEc5b2$) yields expected results as well. The system can sustain even more failures and transmit even more data, however without failing. As Figure 7.9(b) shows the system recovers from notable failures throughout the whole second (i.e., $D \simeq 0kb$ per 30 *sec*) minute, in the fourth

minute (i.e., $D \simeq 20kb$ per 30 *sec*) and in the fifth minute (i.e., $D \simeq 60kb$ per 30 *sec*) without failing completely. In doing so the system manages in this setting to transmit the biggest amount of data (i.e., $D \simeq 1mb$ after six and a half minutes of execution time) as compared to all the other experiments. We attribute this outcome also to increased checkpoint frequency.

Note, that all conducted experiments returned data streams which are fully consistent. That is, in case data items were registered at the Sink service instance they were in the expected order with no gaps between them.

Based on the shown experiments we conclude the redundancy extension of the passive-standby technique to be performing as expected and thus beneficial. Simply put, redundant passive-standby improves the reliability of the system. The more backups nodes the assign the continuous data flow the more robust the system is to simultaneous node failures. This is even reflected in an increased overall data throughput by the system.

7.3 Evaluation of the Self-optimizing Execution

In this section we evaluate the qualitative gain of our work with regard to workflow instance throughput enhancement via a series of experiments. Thereby, we exploit the execution environment, in terms common configuration characteristics, of the baseline system so as to apply our concepts on top of it. Since the throughput of workflow instances is characteristic for the control flow we provide only experiments on the self-optimizing solutions for the control flow.

7.3.1 Safety-Ring Elasticity

As described in Section 5.2 the main approach to throughput self-optimization lies in the dynamic reconfiguration of the environment with service types at underutilized nodes. To this end we evaluate the dynamic deployment of application service types only. Given the outcomes of self-healing experiments in the previous section it is obvious that configuring the system service, i.e., Safety-Ring, to different settings does not significantly affect the throughput. Rather, in most situations the main cause for throughput degradation lies in bottleneck application service types. The SR-nodes evenly distribute their load and to not drag down the throughput. In tun, the bottleneck application service instances have queue pending workflow instances at runtime and thus drag down the throughput. Hence, reconfiguration of the environment with regard to application service types is the only subject to evaluation in the remainder of this section.

7.3.2 Evaluations of the Dynamic Service Deployment

In this experiment set we will evaluate the system behavior for the execution output metric P, with the enabled dynamic deployment of application service types. As a baseline we will use basic system configurations that are enabled with the Safety-Ring. As

observed in previous experiments SR-nodes enforce reliability and should always be present in the system. Moreover, when subjected to computationally intensive service instances they also tend to affect the control flow throughput. In separate evaluation runs we will configure the system to additionally contain 50% or 100% of SR-nodes out of all nodes N, which are then individually subjected to dynamic deployment of application service types. Mobile and faulty nodes are omitted from this experiments set. The complete list of all evaluated experimental system configurations is shown in Table 7.8.

Table 7.8: System configurations for dynamic service type evaluations

Experiment Id	Number of nodes (N)	Number of Safety Ring nodes (S)	Service type self-configuring (d)
DPSRn11s5d0	11	5	no
DPSRn11s11d0	11	5	no
DPSRn11s5d1	11	5	yes
DPSRn11s5d1	11	5	yes
DPSRn20s10d0	11	5	no
DPSRn20s20d0	11	5	no
DPSRn20s10d1	11	5	yes
DPSRn20s20d1	11	5	yes

The threshold evaluation parameters for the optimal forwarding node ranks have been set as described in Table 7.9.

Table 7.9: Threshold values for dynamic service type evaluations

Threshold identifier	Threshold value
Low rank threshold – tsh_{Rmin}	0.5
Low rank evaluation window size – ω_{lr}	5 *sec*
Service instance instantiation threshold value – tsh_{lr}	3
High rank threshold – tsh_{Rmax}	0.95
High rank evaluation window size – ω_{lr}	5 *sec*
Service instance deactivation threshold value – tsh_{lr}	0.25

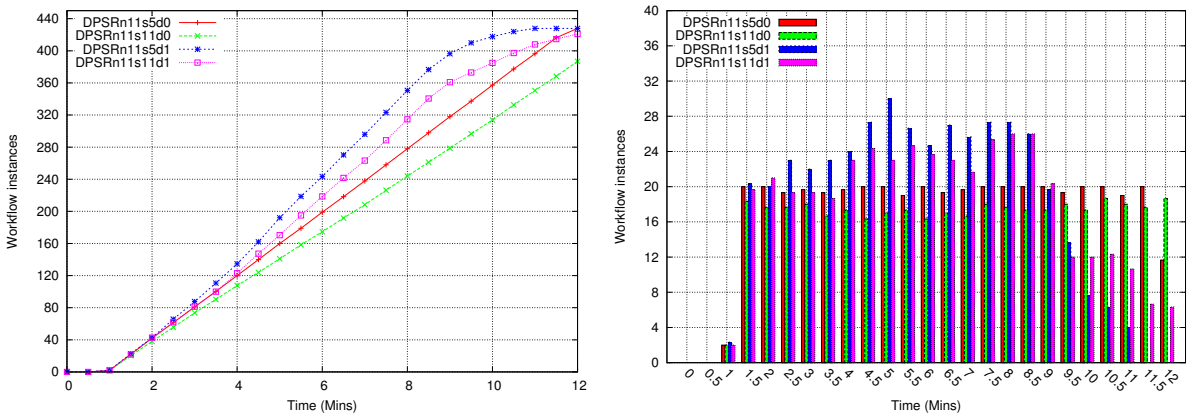
Expected Results

The expectation of our experiments towards the dynamic deployment of application service types will improve the throughput of the control flow of workflow instances. In

particular for the 11 node setting we expect that the dynamic addition of service types should yield better results. The computationally intensive service types *F* and *G* have been identified as bottlenecks, thus the instantiation of service instances should help the system to offload workload to them. This should in turn notably improve *P* for both type of SR-node configurations. In the full SR-node configuration we expect that the additional redundancy of application service types to potential overload some of the SR-nodes and as a consequence feature a lower *P* performance as compared to the 5 SR-node configuration. With the upscaling of the system to 20 we do not expect a significant benefit from dynamic deployment of service types. The reason is that the redundancy of application service types is already grater than one and load will be balanced among the existing service instances from the beginning. The effects of the dynamic deployment should manifest themselves at the end of the execution time since the existing service instances have been overladed first before new ones can be spawned.

Evaluation Results

Figure 7.10 depicts the evaluation results of dynamic deployment enabled system performance with regard to *P* of 11 nodes as compared to the basic Safety-Ring configuration. Precisely, Figure 7.10(a), shows the number of totally finished workflow instances per 30 *sec*. Figure 7.5(b), shows the number of finished workflow instances per per 30 *sec*.



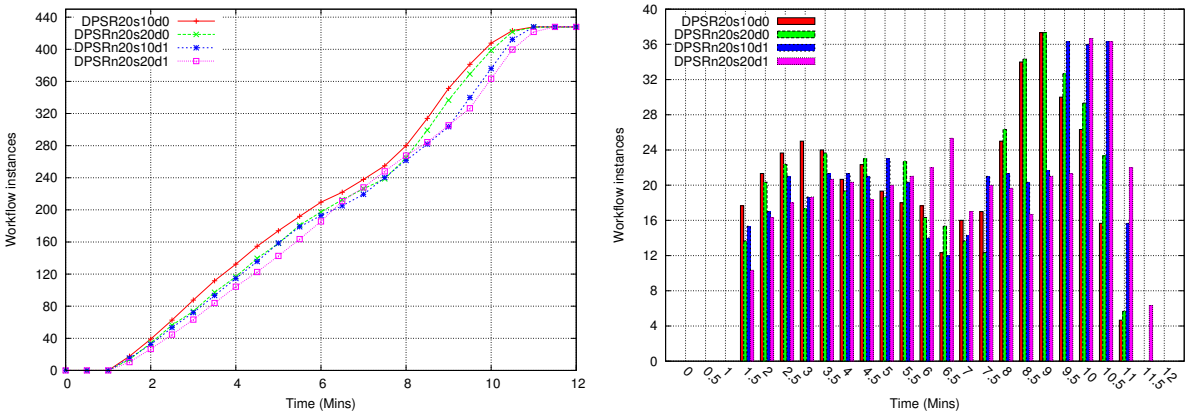
(a) Aggregated workflow instances that have finished

(b) Finished workflow instances per minute

Figure 7.10: Workflow instance execution throughput for 11 nodes with dynamic service deployments

Figure 7.11 depicts the evaluation results of dynamic deployment enabled system performance with regard to *P* of 20 nodes as compared to the basic Safety-Ring configuration. Precisely, Figure 7.11(a), shows the number of totally finished workflow instances per 30 *sec*. Figure 7.11(b), shows the number of finished workflow instances per 30 *sec*.

- *11 node setting* – When it comes to dynamic deployment of application service types we can observe from Figure 7.10 that the system performs as expected.



(a) Aggregated workflow instances that have finished

(b) Finished workflow instances per minute

Figure 7.11: Workflow instance execution throughput for 20 nodes with dynamic service deployments

Compared with the plain Safety-Ring configurations (i.e., $DPSRn11s5d0$ and $DPSRn11s11d0$), which serve in this context as baseline, the throughput strikingly improves for the dynamically deploying configurations (i.e., $DPSRn11s5d1$ and $DPSRn11s11d1$). Already for the five SR-node configuration (i.e., $DPSRn11s5d1$) P improves after just one minute of execution time and continuously increases in the course of the whole execution. Thereby, the average throughput per 30 *sec* increases to around 27 workflow instances. Consequently, the execution of all workflow instances finishes after ten minutes of execution time, which is one minute faster as compared to the plain Safety-Ring ($DPSRn11s5d0$) configuration.

When all nodes are equipped with the Safety-Ring service (i.e., $DPSRn11s11d1$) dynamic deployment helps to improve throughput as well. Although one minute later than in the five SR-node configuration $DPSRn11s5d1$ in $DPSRn11s11d1$ throughput increases to $P \simeq 24$ per 30 *sec* after two minutes of execution time. In the long run this yields about 420 workflow instances which is about 30 instances better than the plain SR-node configuration (i.e., $DPSRn11s11d0$) for the same timespan. Compared with $DPSRn11s5d1$ we believe that overall throughput performance of $DPSRn11s11d1$ is lower for the same reason as $DPSRn11s11d0$ is lower than $DPSRn11s5d0$. Namely, with the instantiation of service instances of possibly high CPU resource requirements (i.e., service types F and G) at SR-nodes an overload at the nodes is caused which consequently slows down the control flow throughput.

- *20 node setting* – Upscaling the system yields somehow unexpected results as shown in Figure 7.11. In this setting, the system does not benefit from dynamic service deployments (i.e., $DPSRn20s10d1$ and $DPSRn20s20d1$) whatsoever, even worse P slightly suffers as compared to the plain SR-node (i.e., $DPSRn20s10d1$ and $DPSRn20s20d1$) settings. Both dynamic service deployments configurations (i.e., $DPSRn20s10d1$ and $DPSRn20s20d1$) feature most of the time a slightly lower

throughput performance. Only during the catch-up time $DPSRn20s10d1$ and $DPSRn20s20d1$ feature a higher performance, but by then the plain Safety-Ring configurations (i.e., $DPSRn20s10d0$ and $DPSRn20s20d0$) have already processed most of their workflow instances. However, dynamic deployment does harm throughput in general as all configurations finish about at the same time,i.e., after the designated ten minutes of execution time. Only $DPSRn20s20d0$ takes 30 seconds more as compared to the others. The only visible benefit that can be attributed to dynamic deployment lies with the configuration $DPSRn20s20d0$ for which the common performance gap at the sixth minute of execution time is bridged with a rather steady throughput of $P \simeq 20$ per 30 *sec*.

The unexpected lack of benefit we attribute to the already existing redundancy in the system w.r.t. service instance numbers. Given this redundancy, instantiation of new service instances is likely applied later in time such that the throughput cannot be visibly improved any more. In the meantime however the *Service Balancer* component constantly consumes local resources in assessing the forwarding node ranks w.r.t. to activation and deactivation decisions, thus causing the slight performance degradation.

Based on the shown experiments we conclude the dynamic deployment of service instances to be performing as expected in general. For small node environments that feature bottleneck service instances the gain in performance is substantial. Whereas for upscaled systems that do not feature any bottlenecks the benefits are either not existing or negligible.

8

Related Work

In this chapter we provide related work in various fields of distributed workflow management. We structure this chapter in three sections such that each one of them addresses related work with regard to the concepts elaborated in the Chapters 3, 4 and 5. The first section discusses the related systems, in terms of distributed management of data, as discussed in Chapter 3. The second section discusses related work for distributed execution of workflows. The third and final section discusses related mechanisms for fault-tolerance of service types and dynamic configuration.

8.1 Distributed Data Management Systems

This section reviews related work addressing data stores for the management of huge quantities of data. Recap, the Chord distributed hash table is the main building block of the Safety-Ring service so as to provide a scalable management of workflow execution metadata.

When it comes to scalable storage and retrieval of data a plethora of solutions have been proposed as a part of the *NoSql* [GHTC13] (*Not Only SQL*) movement in the last years: *CassandraDB* [LM10], *SimpleDB* [Hab10], *DynamoDB* [DHJ+07], *MegastoreDB*, [BBC+11], *BigTable* [CDG+08], *MemcacheDB* [Fit04], *VoldermortDB* [SKG+12], *Pnuts* [CRS+08], *OpenDHT* [RGG+05] etc. According to [Edl15] there are over 150 *NoSQL* data stores as to date.

Although an abundance of NoSQL data stores exists most of them are designed from the bottom up with the same underlying goal of providing a horizontally scalable, highly available, and extremely reliable data stores. Such are the business critical requirements of their respective creators – e.g., *Amazon, Facebook, Google, LinkedIn Yahoo* etc. Just as in the case of the Safety-Ring scalability of the aforementioned data stores is essentially achieved by means of uniform key range partitioning or consistent hashing on top of a key-value related data model. However, as the *CAP* theorem [GL02] mandates the aforementioned characteristics of scalable availability and reliability have to be payed with a price. That price is the sacrificed consistency of the data they manage.

All of the mentioned *NoSQL* data stores feature only relaxed consistency semantics (i.e., eventual consistency) for the sake of superior availability.

As correct workflow instance recovery cannot be guaranteed from possibly inconsistent back-up data, consistency is a first class citizen in Safety-Ring. In the approaches of *Spinnaker*[RST11], *Scatter*[GBK+11], and *Spanner*[CDE+12] strong consistency semantics are not for the trade as well. All of these approaches leverage Paxos for the enforcement of consistency of the managed data inside groups of preselected peers.

Spinnaker focuses on availability and consistency of data, thus offering competitive performance, in terms of concurrent read/write operations, as compared to the eventually consistent systems. Precisely, *Spinnaker* leverages an optimized version of *Multi-Paxos* [Lam98] for data replication on top of the *Zookeeper* [HKJR10] distributed peer coordination service. In line with the CAP theorem, *Spinnaker* thus trades for partition tolerance by assuming its application in a single data center only, where the probability of network partitioning is very low. This in turn limits the scalability of *Spinnaker* to one data center only. *Spanner* offers a similar approach, however it extends the data store beyond a single data center by providing low-level implementation optimizations in integrating concurrency control, Paxos replication and 2*PC*. Moreover, it optimizes consistency w.r.t. transactional read operations by maintaining multiple versions of the same data items and offering them to the clients at points in time which are considered to be consistent. *Scatter* goes more in the same direction as the Safety-Ring by providing strong consistency for Peer-to-Peer data management systems. Precisely, Scatter provides Paxos based replication inside groups. In turn, these peer groups are subjected to a Peer-to-Peer structural organization (e.g., a Chord ring) that are coordinated in a consistent fashion among themselves by means of 2PC transactions. This yields a fully consistent distributed data management system, however at the price of significantly reduced data operations throughput, i.e., availability.

In contrast to aforementioned strongly consistent and scalable data stores, Safety Ring trades for availability, as defined by the CAP theorem, by leveraging the fault tolerant Paxos commit in conjunction with the Symmetric replication scheme. Those two building blocks are specifically tailored for reliability of the underlying infrastructure and are thus neglecting availability of the system as the evaluations of Section 7.2 suggest. We believe however that the underlying scalable infrastructure (i.e., Chord ring) and the additional data access optimizations (i.e., Compass) help to somehow mitigate the availability issue. The Safety-Ring data store leveraged for the storage of workflow instances is a slightly optimized version of the Paxos commit data store as elaborated in [SRHS10] by slightly reducing the number of exchanged messages.

Other distributed data management systems that imply strong consistency semantics such as the distributed file systems *GFS* [GGL03], and *Apache Hadoop* [BGS+11] are also designed to be operated in more stable execution environments – a single data center. Distributed file systems feature master nodes that coordinate read/write operations for a scalable amount of data resource nodes. Thereby, the write operations are assumed to be append-only, i.e., once created no data item is changed ever. This goes along lines of the application scenario of the DFSs, which are big data centers that are constantly extended with new computational resources. The master nodes are assumed to be sta-

ble and often have to be manually reconciled in the event of failure which is not possible for Peer-to-Peer environments.

8.2 Distributed Workflow Management

Although workflow management has been discussed intensively, in terms of the traditional – centralized – execution model, in Section 1.2 along with the problems associated to it in Section 1.3, this section is completely devoted to related workflow engines that apply a distributed execution model. Before we start describing them we briefly describe how workflows can be formally defined as well.

8.2.1 Related Workflow Formalism

To describe the dynamic behavior of a workflow instance at runtime (i.e., its state changes) formal state transition systems have to be leveraged There are several formalisms such as BPEL4WS [KMW03], BPMN [WG08, VDAHW03, BCPV04], etc. aiming to describe the structural semantics of a workflow definition by means of a flowcharts [NS73]. However, flowchart based approaches all lack the formal mathematical semantics, in terms of state transition, needed to formally describe workflow instance behavior at runtime. The *Petri-Nets* [Mur89, Aal98, Rei13, VDAS11] offers a well-established mathematical model for the simultaneous representation of workflow definition characteristics in static (i.e., their structure) and dynamic (i.e., their control flow) terms.

In using Petri-Nets, the structure of a workflow definition is defined to be a directed bipartite graph that is composed of nodes and relations among them that reflect control flow dependencies. Thereby, two types of nodes are distinguished, namely *Places (P)* and *Transitions (T)*, that are connected by directed edges, i.e., *Arcs*. To denote state in a Petri-Net graph a certain number of tokens is assigned to each place node. The given distribution of tokens at all places at any point in time thus represents the current state of the Petri-Net graph from a control flow point of view. The distribution of tokens is referred to as *marking* of place nodes.

Workflow definition structures can be represented by means of Petri-Net graphs in exploiting the transitions for representing the activities and in exploiting the places for representing the control flow precedence order relations. In doing so, the control flow semantics have to be upheld by directing the Petri-Net arcs among the nodes so as to coincide with the precedence order relations. Simply put, a directed place can be interjected only between two transitions whose mapped activities precede each other due to a control flow precedence order.

During execution of a workflow instance its state is subject to change. To represent the change in execution state, Petri-Nets perform redistribution of the tokens among the place nodes. A transition that features at all of its incoming places tokes is referred to in Petri-Net terminology as *enabled* and can invoke its mapping service instance. An enabled transition consequently fires, i.e., it consumes a token from each of its input places and produces token at each of its output places, and thus marks its execution

state advancement. The subsequent firing of enabled transitions represents the Petri-Net based state transitions.

The Petri-Net model is perfectly suitable for discrete state transitional semantics such as in the case of the control flow. However, the Petri-Net model does not address in its basic form neither the subject of data transfer among the transitions nor the continuous liveliness of activities, such as in the case of the data flow. Although the association of data to the control flow can be accommodated with simple extensions of the basic model by means of *colored Petri-Nets* [Jen96], continuous service types are not as simple. Colored Petri-Nets imply additional color tokens that can be used to encode arbitrary complex types of data. To represent continuous transitions the time dimension has to be formally introduced into the basic model by means of *Timed Petri-Nets* [Zub80, Bow00] which further entail complex synchronization constructs among the transitions. Timed Petri-Nets are due to their complexity are over excessive for our simple workflow model and thus outside the scope of this thesis.

8.2.2 Distributed Workflow Engines

One of the first approaches to distribution of workflow execution is suggested in *Exotica/FMQM*[AMG+95] where the benefits of scalability are identified early on. Exotica/FMQM goes along the same lines as OSIRIS in migrating workflow instance among service instances at runtime. In doing so Exotica/FMQM necessitates the provisioning of workflow definitions at service instances before the execution of its instance even starts. This implies that the individual activities of the workflow definition are decomposed and their hosing nodes supplied with the definitions. Moreover, the invariant data of the data flow has to be per-distributed as well so as to be able to invoke the corresponding activities. By distributing invariant data items to specific nodes, runtime control flow among the instances is implicitly determined at the compilation (i.e., build-time) of the workflow definition.

To overcome this rigid approach to late binding *AltaVista* [BGE99] introduces dynamic replication of data among the service instances at runtime. AltaVista features replication of data flow data items among service instances at runtime through a centralized (i.e., master) replication server. Although data propagation via the master server features simple replication optimizations (i.e., autonomous slave replication servers) it turns out to be a permanence bottleneck eventually.

Exotica/FMQM and AltaVista are quite different from OSIRIS, even though is has to analyze the workflow definitions at compile-time so as to obtain the metadata. OSIRIS features a dynamic data flow by sending the data along with the workflow instance and selecting the service instances dynamically at runtime based on locally available metadata.

Agent based approaches of *NIÑOS*[LMaJ10], *DartFlow* [CGN97], *AMOR*[BCF+06], *CEKK* [YY07] optimize for the allocation of resources of nodes by avoiding the provisioning of workflow definitions to all nodes. That is they feature smart agents which posses the entire workflow definitions and resolve their execution paths dynamically at runtime. The agents also convey the execution data flow along with them. In doing so agents can also perform semantic path detection (i.e., AMOR and CEKK) for recovery

purposes. For this to work AMOR relies on metadata repositories on service instance availability just as OSIRIS at migration time.

Whereas NIÑOS leverages the *PADRES* [FJLM05] distributed publish/subscribe system for the coordination of agents. That is, activity agents located at free service instances subscribe to a workflow instance agent, i.e., *process* agent – as referred in NIÑOS terminology. In turn, the workflow instance agent, which is also a pub/sub client, coordinates the execution by publishing the workflow instance to the subscribed activity agents along the lines of the control flow precedence order. The PADRES framework carries out load balancing and content-based routing of agent subscriptions through a network overlay of message brokers in a distributed fashion, i.e., by means of *Minimum Spanning Trees* [CLRS09].

JOpera [PA04, HPA05] also features a scalable workflow engine, which can be horizontally and vertically (i.e., by adding new threads to it) distributed at nodes. To power JOpera a global workflow instance state and environment configuration state is maintained at a centralized node. This state is however replicated to across all redundant engines at runtime in a Peer-To-Peer fashion such that the centrality of the global repository is alleviated. That is, each workflow engine node locally maintains routing tables which provide it with information on workflow instance state that is located at other nodes. This data however needs to be prefetched at runtime from the other nodes before the invocation of a local activity can take place. In contrast to JOpera, configuration state is already existent at the orchestrator nodes at runtime, whereas workflow instance state is not shared through the metadata repository due to the inevitable scalability issue of it. Similar to OSIRIS, JOpera [BP09, BPA06] also supports continuous service types within its workflow structures. However, unlike OSIRIS where discrete invocations are only used to activate streaming service instances, in JOpera arbitrary (interaction) combinations of discrete and continuous service types within the structure of a workflow are possible.

8.3 Self-Healing Workflow Management

This section reviews the aforementioned distributed workflow engines with a particular focus on the robustness to node failures. Moreover, it discusses related solutions to node fault-recovery as applied in the Safety-Ring. Finally, it discuses related reliability solutions in the context of the continuous data flow.

8.3.1 Reliable Control Flow Execution

Already the first distributed workflow engines (i.e., Exotica/FMQM, CEKK) have recognized the importance of fault tolerance mechanisms for node failures. At that time the recovery techniques were based on rather simple solutions,i.e., persistent message queues, that were used to locally store workflow instance state at migration time. This implies that the recovery solutions addressed failures of temporary type only. CEKK took the persistent message queues a step forward by enforcing them by means of 2PC replication to a set of backup nodes.

Since then reliability of distributed workflow management has come a long way. The works of [Yu10] extend CEKK with *scope* managers that oversee distributed workflow instance migration by storing backup data. In contrast to Safety-Ring the scope managers are centralized nodes that do not scale well.

In [Ye06] redundant service instances are equipped with agents that perform monitoring and recovery. Although this approach does not rely on centralized nodes, it again suffers from scalability issues. In the event of active service instance failure, a non-scalable leader election algorithm has to be conducted among the redundant recovery agents. On the contrary, in Safety-Ring the service instance is effectively recovered by a SR-node, and a SR-node is recovered by its successor.

The solutions of [KCS10, JKG09] suggest rollback recovery in conjunction to state checkpointing for mobile agents. These approaches rather focus of optimizing the checkpointing frequency for the sake of improved execution performance and do not consider advanced network failure scenarios, i.e., the backup stores are considered to be idle.

Musasabi [AUS+09] proposes a similar approach to achieve reliability in Peer-To-Peer systems. Namely, Musasabi proposes virtual peers, that are composed of multiple redundant ones, that share process state by means of Paxos replication. However, Musasabi structures the peers by means of the *Skip Graphs* [AS07] network overlay such that they are resources they are mapping are close to each other. Given a high load such an approach would feature an unbalanced load among the peers, as compared to Safety-Ring where the SR-nodes are evenly loaded.

In NIÑOS the workflow instance agent subscribes to the message flow and can based on this deduce failure of activity agents. Since control of the workflow instance is centralized by the manager so is the recovery. JOpera achieves fault-tolerance by periodically checking the configuration (i.e., routing tables) state among the nodes by pinging the connected nodes. In case nodes fail the required state for invocation of local activities is obtained at the global repository. For this to function each workflow engine has to write its updates to the workflow instance state both to its connected nodes, obtained through the local configuration state, and the global repository. Consistency of the replication process is not elaborated in JOpera and the centrality of the global repository still persists.

8.3.2 Reliable Data Flow Execution

When it comes to reliability of the continuous data flow distributed data stream engines are proposed in literature. According to the works of *Aurora* [HBR+05] three general approaches to data streaming reliability are suggested. Namely, the recovery techniques of *active-standby*, *passive-standby*, and as an alternative to *passive-standby* – the *upstream-backup* are suggested. Recap, active-standby features completely redundant data flows at different service instances. Whereas passive standby features pairs of service instances that share the streaming state by means of checkpointing, such that one service instances is active and periodically sends state checkpoints to the passive service instance for backup purposes. In upstream-backup checkpoints are not sent to a preselected backup node but rather to the upstream service instances. Aurora recom-

mends the upstream-backup approach as it features the lower backup overhead than active-standby but better recovery performance than passive-standby.

Aiming for a much higher throughput, in the event of a failure, the *Borealis* [BBMS05] streaming engine opts for the active-standby approach. To achieve immediate recovery Borealis accepts inconsistent data streams that feature gaps, i.e., missing data items. The inconsistencies are reconciled eventually by temporarily filling the gaps with intermediate data items and replacing them with the actual ones as soon as they are available.

Although passive-standby is the most commonly used approach, improved reliability in the context of multiple backup nodes, is not very much considered in literature but before the works of [HXZ07]. In this approach the state of the streaming service instance is split into multiple logical units and then replicated to different backup nodes. By splitting the state across multiple nodes, it can be retrieved much faster at recovery time by means of parallel queries to the backup nodes. Moreover, checkpoints are optimally scheduled in time such that the recovery overhead is minimal. On the other hand, state splitting implies that the consistency of the split sub-states at the remote nodes can only be guaranteed if the stream processing is deferred (i.e., stopped) until a checkpoint is successfully finished. That is, in [HXZ07] synchronous checkpointing is featured. Depending on how fine-grained the redundant state should be, logical units can be dynamically allocated so as to join/split the streaming instance.

As compared to [HXZ07], OSIRIS-rSE applies asynchronous checkpointing and thus does not affect stream processing. Assuming that state splitting is possible (or allowed) due to the nature of the data type that is used to represent it, in OSIRIS-rSE splitting is not necessary at all. The ECOC algorithm of OSIRIS-rSE substantially minimizes the checkpointing state volume (i.e., omits the input/output buffers) and is thus efficient by default.

A similar approach to [HXZ07] is followed in *D-Stream* [ZDL+12], in terms of redundant checkpoints. The main difference is that in D-Stream metadata on data stream is check-pointed instead of actual service instance state. This metadata, also referred to as *Resilient distributed datasets – RDD* [ZCD+12], is less voluminous and can be used to efficiently reconstruct the original data stream at runtime. The RDDs can be arbitrarily fine grained so as to support efficient recovery as well, but they need to be replicated in a synchronous fashion.

Asynchronous checkpointing of service instance state along with strong consistency semantics (just as in our case) is achieved in *S-guard* [KBG08]. S-guard is an extension of the [HXZ07] project aimed for huge data center application scenarios and features the same state splitting characteristics. Moreover, it introduces a custom memory management unit that allows for asynchronous state checkpointing at a stable storage such as a reliable distributed file system, i.e., Hadoop or GFS. In turn, the DFS provides the strong consistency semantics of the checkpoints which are executed asynchronously by the streaming instance. The checkpointing coordinator is made fault-tolerant, just as in our case, by using rollback recovery on it in the event of its failure.

Even though S-guard checkpointing concept bears a certain similarity to OSIRIS-rSE it cannot be applied in our case. S-guard offloads the biggest difficulty of the data flow reliability, i.e., consistency of the backed-up state, to stable storages. These in turn,

assume unlimited resources so as to achieve their goals, which unrealistic in heterogeneous Peer-To-Peer environments.

Finally, in [BP14] alternative approaches to data stream reliability are proposed which are based on centralized and stable monitoring components in conjunction with service instance migration. Such approaches are mainly intended for more controlled execution environments (e.g., Cloud data centers) and thus not applicable to heterogeneous P2P environments of OSIRIS.

8.4 Self-Optimizing Workflow Management

This section reviews distributed workflow management with regard to service type hot deployment. Moreover, it reviews related Chord optimization approaches, that are, just as Compass, primarily based on distance vector routing techniques.

8.4.1 Chord Optimizations

Since the beginnings of Chord, the need to enhance its core features has been widely accepted, especially when it comes to applying Chord to dynamic, unreliable and/or heterogeneous environments such as mobile ad-hoc networks,e.g., *MANETs*.

The first Chord enhancement techniques, are also based on the idea of assessing latency among nodes so as to minimize routing times. The approaches based on *Proximity Neighbour Selection(PNS)* [CDCR02, DLS$^+$04] and *Proximity Route Selection (PRS)* [MWSP08] are the most prominent ones. Thereby, the PNS approach of [DLS$^+$04] on average outperforms the others with regard to latency and network resource consumption. However, PNS-based approaches are in general based on latency probing of random candidate peers [MJB05] for the construction of the Finger Tables. This way, the standard Chord algorithm is significantly altered, and even more, not all possible nodes are considered for minimal latency paths. Compass instead evaluates all nodes for the latency optimal paths and complements the Finger Tables only, it does not replace them. Moreover, PNS approaches are less applicable to MANETs due to the volatile nature of such environments. For instance, MANETs feature a high and varying data packet loss rate and significant delays due to collisions and interference among nodes.

Nevertheless, [SGF02] suggested the similarities of Peer-To-Peer environments and MANETs, expressing the need to combine both, due to the same problems at hand when it comes to self-organization, scalability and robustness. Advocated solutions in [SGF02] include, just like in Compass, integrations of Chord with proven networking protocols such as Distance Vector Routing – DSDV [PB94, Bla00].

The Compass Table convergence process resembles the Distance Vector Routing protocol which is also founded on Bellman-Ford algorithm for shortest paths computation. DSDV bases its path destinations computations on nodes and not on key identifiers like Compass, which are eventually associated to nodes. Moreover, based on the concrete implementation [Hed88], DVS protocols may posses different distance metrics than latency, for example minimal number of hops again. Finally, in the DVS routing table re-

trieval performance is dependent on the actual node topology which might be in some cases not scalable as it always is in a ring topology of Chord.

Guided by the aspiration of optimal and scalable content-centric node routing in MANETs, the various approaches of [FDKC06, MJ07, LWW10, FY09, HGRW06, ZS06, PDH04] integrate DHT substrates, among others Chord, on top of networking layers. Commonly, in the works of [MJ07, LWW10, FY09, HGRW06] the routing technique of the network layer utilized for Chord hops is AODV [PBRD03], which is an implementation of distance vector routing. For these approaches the Chord Finger Table is consulted first so as to find the next forwarding node. Afterwards the AODV routing tables are consulted so as to find the optimal physical routes to the next forwarding node. Compass works the other way around, by consulting the Compass Table first, complemented on top of the default Chord. This way, latency optimal paths are utilized first at the expense of additional hops as compared to default Chord. In the case of non-existing optimal paths, our approach always resorts to default Finger Tables.

In [MJ07, FY09] complex *random landmarking* [WZS04] algorithms are used for the construction of Finger Tables that comprise physically proximate nodes, which are unnecessary in our case as physically proximate nodes emerge from latency optimal paths. Random landmarking differs from the Lipschitz [JL84] approach in assigning the landmark nodes dynamically at runtime, whereas with Lipschitz the landmarks are static and predetermined by expert users.

In terms of network traffic, [MJ07] and [LWW10] mainly focus on minimizing it by reducing the number of data exchanging nodes to only two, i.e., the predecessor and the successor. The rest of the ring is learned by overhearing/piggybacking traversing messages which in the long run yields a much slower optimal path convergence as compared to the Compass Tables. Mo rover, in [HGRW06] network traffic reduction is the most important topic and it is achieved by means of multicast messaging of AODV.

An alternative to optimal latency paths that is similar to distance vector routing lies with *Dynamic Source Routing (DSR)* [JM96] as applied by [FDKC06]. DSR finds the latency optimal paths reactively, i.e., during route discovery at runtime. In turn, the optimal paths (containing all optimal traversal nodes) have to be cached at intermediate nodes in the network. Such an approach is characterized by limited knowledge of optimal paths (i.e., only the frequently used node paths) and high overhead (i.e.,, the complete optimal path per destination node has to be stored) at the intermediate nodes. Moreover, convergence of updated paths is slow given a high ring churn rate.

Self-Optimizing Workflow Engines

When it comes to optimization of the distributed workflow engine, in terms of execution performance, the main body of work focuses on optimized late-binding. Already the first distributed workflow engines approaches of [BD00] introduce load balancing among the service instances at runtime. In doing so, the selection of service instances is refined by taking into account the dynamic communication costs. Similar to OSIRIS, in the *ADULA* [MB11, MB09] framework late-binding of optimal service instances, in the context of BPEL[1] processes, is supported by a performance monitoring mechanism.

[1]BPEL processes semantically correspond to OSIRIS workflows.

Thereby, ADULA ensures maintenance of process performance through automated detection of service failures and SLA violations, diagnosis, and repair. The ADULA framework allows for monitoring of process and service performance using lightweight sampling technique and leveraging statistic methods such as Bayesian inference, or statistical hypothesis testing. Although of similar intention, OSIRIS late-binding is not based on SLA enforcement and features simpler statistic methods.

Regarding the reconfiguration of the execution environment, with application and system service types very little work exists in the literature to the best of our knowledge. Not before [LSPF07] the need to dynamically reconfigure workflow engines by means of autonomic computing model has been recognized in the research community. In SOSOA[BBP+11] the general lack of self-optimizing workflow engines bas been identified and put into a broader context, i.e., self-organization of distributed workflow engines.

The only notable approaches to dynamic reconfiguration of the workflow engine are the already mentioned approaches of NIÑOS[GYJ11] and JOpera [HP08]. In a centralized approach, NIÑOS proposes the introduction of a feedback loop controller at each manager agent (i.e., the workflow instance agent) that monitors the load of the operation agents (i.e., activity agents) such that the instantiations of the operation agents can be. That is, depending of the monitored load at the operation agents new ones can be added or underutilized ones removed by the manager agents.

Similarly, JOpera runs its dynamic deployment of workflow engine instances at nodes on a centralized feedback loop controller. That is global metadata repository aggregates the configuration state of the whole engine and devises at runtime optimal configurations with regard to load. These are in turn executed by the local instances of the workflow engine by adding/removing execution threads from/to it. In contrast to NIÑOS, the controller of JOpera is powered by simple policies that allow for sophisticated and timely configurations of the system and overcome the need for its manual configuration (e.g., setting threshold values as in our case) by an expert end user of the system. Moreover, JOpera [PPBB14] redesigns the architecture of traditional workflow execution engines so as to take full advantage of the modern underlying hardware resources, comprising several chip multiprocessors each offering multiple cores and a large shared memory cache. To this end JOpera additionally introduces self-configuration of the engine upon startup in order to optimize the utilization of the available local hardware (i.e., the type and number of available chip multiprocessors) without sacrificing the portability of the engine across different processor micro-architectures. As a result replication used in conjunction with CPU affinity binding techniques help increase execution performance of it.

The dynamic deployment of services in OSIRIS is not as fine-grained as it is in the case of JOpera, i.e., it needs manual configuration at startup. On the other hand, it encompasses application services and it is completely distributed. Each orchestrator node can based on local statistics metadata autonomously configure the environment at runtime. Since the metadata is equally distributed, by the central super peer, and all orchestrator nodes are running the same configuration procedure the should come to the same deployment conclusion. Hence, the dispersion of service types in the environment should be limited and controlled.

9

Conclusions

In this chapter, we conclude this thesis by summarizing the main contributions of our work and discuss directions of future work that can improve our contributions even more.

9.1 Summary

In this thesis we have presented a distributed execution model for workflows that features self-healing and self-optimization characteristics. First we have motivated our work by means of an application scenario (i.e., modern emergency management) that runs in heterogeneous environment for which we have determined a set of requirements on an ideal workflow engine. These include conformance for all types and any combinations of service deployments (i.e., continuous and discrete); scalable distribution of the engine services; reliable execution of the control flow, the data flow and the engine itself; and finally resource conservation by the services of the engine.

To meet some of the requirements of an ideal workflow engine we have laid foundation for this thesis by defining a formal distributed workflow execution model. The formal model clearly defines distributed workflows with regard to their structure, runtime execution behavior and the distributed (i.e., scalable) system that runs them. The introduced distributed execution model already meets the requirements of service conformance, scalability and resource conservation as it is.

However, since the environment in which the engine operates is heterogeneous in its nature node failure situations have been identified that can critically affect distributed execution model. To overcome node failures we have presented the self-healing Safety-Ring system service that provides a novel approach to control flow reliability for active service instance failures. We have discussed how Safety-Ring constructs a scalable and self-organizing node overlay (i.e., SR-nodes) with the purpose of active service instance monitoring and recovery. Since the monitoring node overlay can be used for the recovery of the monitoring nodes themselves, we have asserted the Safety-Ring service to be self-healing. Moreover, we have described in detail the centerpiece of the Safety-Ring which is a scalable, reliable, and consistent data store. The Safety-Ring data store is used

by the monitoring node overlay for the storage of runtime execution state, i.e., workflow instances and monitoring node assignments. To explain the detailed functionality of the data store of Safety-Ring, we have provided theoretical background on the applied concepts of the distributed data management such as distributed hash tables (i.e., Chord), symmetric replication techniques and distributed transactions (i.e., Paxos commit).

Since Safety-Ring assumes rather stable network runtime characteristics, it is not by default designed for heterogeneous environments. To overcome this issue, we have extended the Safety-Ring service with Compass data access protocol. With the introduction of Compass we have highlighted the problems of Chord when applied to mobile environments in which the network characteristics of nodes dynamically change. We have shown how latency optimal paths to each node in the environment can be found and dynamically maintained without affecting the scalability of Chord.

To address the issue of the reliable data flow Safety-Ring is leveraged in the context of discrete service instances. However, data flows of continuous (streaming) service instances have to supported by more resource conservative recovery strategies. In this thesis we have presented how proven techniques for data stream reliability can be extended for redundancy so as to improve robustness of the continuous data flow. Thereby, theoretical background on the most common recovery technique, i.e., passive-standby, has been provided as well. Moreover, we have shown that the increased redundancy does not have to suffer from consistency issues and have provided a consistency protocol (based on 2PC) that is lightweight enough to be applied in heterogeneous environments.

With the Safety-Ring and redundant-passive standby extension we have designed the distributed execution model of workflows for reliability. In this thesis we have gone a step further in improving the throughput performance characteristics of the distributed execution model as well. That is, we have presented a novel approach to dynamic service type deployment inside the execution environment. The presented approach is based on decentralized controllers which autonomously provision the environment service instances with the goals of preventing execution bottlenecks at runtime and unnecessary deployments – from a resource consumption point of view. Since the controllers are equipped at any node in the system and can affect any other node of in system with deployment decisions the distributed execution model is asserted to be self-optimizing.

Moreover, we have implemented Safety-Ring, passive-standby, Compass and the dynamic deployment controllers within the context of the OSIRIS distributed workflow engine. To do so, we have presented the architectural model of OSIRIS and described how the components of it have been used to implement our concepts.

Finally, the OSIRIS implementations have been subjected to a series of experiments which simulate distributed execution in volatile and heterogeneous environments. Based on the evaluations we had come to the conclusion that our concepts have helped to improve the reliability and performance of the distributed execution model. That is, the reliability requirement of the ideal workflow engine has been successfully met. Safety-Ring and redundant passive-standby improved the robustness of the control flow and the data flow to node failures, whereas Compass helped to transfer the reliability benefits to heterogeneous environments. On the other hand, the centralized

controllers improved the throughput of workflow instances, albeit for small node environments only.

9.2 Future Work

When it comes to future research of our work we propose several interesting directions. First of all, the costs of Safety-Ring reliability, reflected in reduced throughput, could be mitigated with the help of batching. Essentially, the number of transactional writes to the Safety-Ring could be reduced if each SR-nodes would aggregate over time concurrent distributed transactions before starting with their processing. Having an overview over a number of concurrent transactions each SR-node could prevent potential conflicts by merging parallel workflow instances beforehand and thus save itself the Paxos commit procedure for writes which are destined to be aborted. For instance, if two parallel activities are trying to migrate to the same join activity the conflict probability of their corresponding transactions is very high. With batching, one of the concurrent transactions could be stopped (or aborted) while the other one could be admitted for Paxos commit processing with the already merged workflow instance state. This way the number of Paxos commit transactions on the system would be reduced and resources (i.e., bandwidth) at the nodes significantly conserved. In general, workflow definitions that are characterized by structures of high parallelism (i.e., very many parallel activities and branches) should benefit from batching as the number of concurrent transactions would be reduced.

The Safety-Ring recovery mechanism features simple re-invocation of the failed service instances. In case the there are substitution service instances to recover the failed ones the Safety-Ring can only abort the workflow instance execution. Therefore, SR-nodes could be equipped for semantic recovery of node failures. Having semantical preference orders at disposal each SR-node could at runtime infer alternative execution paths recover failures by taking them instead.

Another potential research topic could involve the application of more sophisticate machine learning tools for our work. Thereby, the decentralized controllers represent only a first step into research field of artificial intelligence. Currently, all of our self-optimization features (e.g., Safety-Ring elasticity, application service deployment) rely on static threshold values that have to be set by expert users of the workflow engine, i.e., administrators. If the controllers were supported with machine learning tools and more extensive metadata on the execution environment, configurations of them could be learned that improve system throughput without the need for human intervention, i.e., expertise. Distributing such an machine learning approach onto the decentralized controllers in a scalable fashion would certainly pose another interesting research topic by itself.

Since the distributed controllers rely on metadata freshness for their decision making of service deployments we could facilitate their efforts with more more sophisticated metadata propagation. That is, freshness of the metadata could be improved if the global metadata repository was distributed to multiple nodes which are physically closer to subscription nodes. In the context to the modern emergency management sce-

nario the global metadata repository located at the headquarters could be distributed to multiple firefighter trucks. Based physical locality the devices at the fireman could subscribe to nearest firetruck and thus benefit from faster data transmission. In turn, the trucks would be connected to the headquarters via more powerful data transmission lines and in conjunction with sophisticated replication algorithms prefetch parts of metadata that would be of interest to the firemen. At the heart of this approach would be a distributed and self-organizing tree structure that propagates data in a pub/sub fashion. At the leaf level of such a tree we would repository nodes that maintain and publish the metadata only on the subscribed nodes of orchestration service type. Whereas at the inner level we would have higher level repository nodes that aggregate more metadata (i.e., metadata that reflects all subscribed nodes down the tree) and publish preselected parts of it to repository nodes at the lower levels. At the root level we would have the global repository node that aggregates all of the metadata for all nodes in the environment. Such a repository nodes tree structure would have to dynamically reorganize itself (i.e., self-organize), in terms of inner repository node numbers (levels), such that the overall metadata throughput is maximized.

A

List of Acronyms and Symbols

In the following we list a table that summarizes all acronyms and symbols used in chapters 3, 4 and 5. We additionally provide a brief description and the location of each acronym or symbol.

Acronym/ Symbol	Explanation	Location
$P2P$	Peer-to-Peer	Chapter 3
KI	Key identifier space set	Section 3.1
f_h	Key set parametrized hash function	Section 3.1
f_n	Node set parametrized hash function	Section 3.1
K	Data items key set	Section 3.1, Definition 3.1
N	Nodes set	Section 3.1, Definition 3.1
$assign$	Key set parametrized node assignment function	Section 3.1, Definition 3.1
KIP	Key identifier space partition set	Section 3.1, Definition 3.2
$resp$	Key identifier node responsibility function	Section 3.1, Definition 3.3
$succ$	Ring topology successor assignment function	Section 3.1, Definition 3.4
fn	Finger Table node	Section 3.1, Definition 3.6
FT	Finger Table	Section 3.1, Definition 3.7
Φ	Common data alphabet	Section 3.2, Definition 3.8
DI	Data item set	Section 3.2, Definition 3.8

DHT	Distributed hash table	Section 3.2, Definition 3.9
R	Replication factor set	Section 3.2
R	Hash function set	Section 3.2.1
SL	Successors set	Section 3.2.2
EC	Symmetric replication equivalence class set	Section 3.2.3, Definition 3.10
$symm$	Key identifier association function of an equivalence class	Section 3.2.3, Definition 3.10
DHT_s	Symmetric replication enabled DHT	Section 3.2.3, Definition 3.11
T	Timestamps set	Section 3.3
VH	DHT version history set	Section 3.3
$item$	Transaction item	Section 3.3, Definition 3.13
DTX	Distributed transaction set	Section 3.3, Definition 3.14
DHT_{tx}	Distributed transaction enabled DHT	Section 3.3, Definition 3.15
$2PC$	Two phase commit	Section 3.3.2

Table A.1: Chapter 3 explanations of acronyms and symbols

Acronym/ Symbol	Explanation	Location
s_d	Discrete service type	Section 4.1, Definition 4.1
f_p	Business logic function of s_d	Section 4.1, Definition 4.1
DST	Data stream	Section 4.1, Definition 4.2
SS	Internal state set of streaming instances	Section 4.1
s_c	Continuous service type	Section 4.1, Definition 4.3
f_c	Business logic function of s_c	Section 4.1, Definition 4.3
S	Service type set	Section 4.1, Definition 4.4
SI	Service instances set	Section 4.1, Definition 4.5
WF	Workflow definition set	Section 4.1, Definition 4.6

A	Workflow activity set	Section 4.1, Definition 4.6
$\prec_{cf}$	Workflow precedence order set for the control flow	Section 4.1, Definition 4.6
$\prec_{df}$	Workflow precedence order set for the data flow	Section 4.1, Definition 4.6
a_s	Workflow definition start activity	Section 4.1, Definition 4.7
A_e	Workflow definition end activity set	Section 4.1, Definition 4.7
WI	Workflow instance set	Section 4.1, Definition 4.8
A_c	Completed activity set of a workflow instance	Section 4.1, Definition 4.8
A_s	Subsequent activity set of a workflow instance	Section 4.1, Definition 4.8
A_a	Current activity set of a workflow instance	Section 4.1, Definition 4.8
SI_a	Set of currently active service stances of a workflow instance	Section 4.1, Definition 4.8
SI_c	Set of traversed service stances of a workflow instance	Section 4.1, Definition 4.8
WI_t	Workflow instance version history set	Section 4.1
f_{st}	Streaming service instances state representation function	Section 4.1.2
s_o	Orchestration service	Section 4.2
SI_o	Orchestration service instance set	Section 4.2
H	Service host metadata set	Section 4.2, Definition 4.10
SUB	Metadata subscriptions set	Section 4.2, Definition 4.11
ρ	Global metadata repository	Section 4.2, Definition 4.12
MD	Orchestrator metadata set	Section 4.2, Definition 4.13
J	Join service instance metadata set	Section 4.2.2, Definition 4.14

Table A.2: Chapter 4 explanations of acronyms and symbols

Acronym/ Symbol	Explanation	Location
s_{sr}	Safety-Ring service type	Section 5.1.1
SI_{sr}	Safety-Ring service instance set	Section 5.1.1
f_{mt}	SR-node monitoring assignment function	Section 5.1.1
$\prec_{sr}$	SR-node successor relations set	Section 5.1.1
f_{wd}	Workflow instance - data item mapping function	Section 5.1.1
f_{dw}	Data item - workflow instance mapping function	Section 5.1.1
SRS	Safety-Ring	Section 5.1.1, Definition 5.1
CT	Compass Table set	Section 5.1.2, Definition 5.2
f_{lat}	Note latency assignment function	Section 5.1.2
CH	Checkpoint set of redundant passive-standby	Section 5.1.3, Definition 5.3
CT_{st}	Stability category set	Section 5.2.1
TSH_{st}	Stability threshold set	Section 5.2.1
ST	Stability metadata set	Section 5.2.1, Definition 5.4
M	Mobility metadata set	Section 5.2.1, Definition 5.5
f_m	Mobility distance function	Section 5.2.1
LMK	Landmark node set	Section 5.2.1
f_{msr}	Landmark latency assignment function	Section 5.2.1
LV	Landmark distance coordinates set	Section 5.2.1
LT	Latency metadata set	Section 5.2.1, Definition 5.6
f_{lt}	Latency distance function	Section 5.2.1
MD_e	Extended orchestrator metadata set	Section 5.2.1, Definition 5.7
$STAT_{sr}$	SR-node load statistic	Section 5.2.2, Definition 5.8
$STAT_{rk}$	Rank statistic	Section 5.2.2, Definition 5.9

Table A.3: Chapter 4 explanations of acronyms and symbols

Bibliography

[Aal98] W.M.P. Van Der Aalst. The Application of Petri Nets to Workflow Management, 1998.

[ACKM10] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[AG10] Nick Antonopoulos and Lee Gillam. *Cloud Computing: Principles, Systems and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[AHK+02] Wil Van Der Aalst, Kees Van Hee, Prof. Dr. Kees, Max Hee, Remmert Remmerts De Vries, Jaap Rigter, Eric Verbeek, and Marc Voorhoeve. Workflow Management: Models, Methods, and Systems, 2002.

[AMG+95] G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A.El Abbadi, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Information Systems Development for Decentralized Organizations*, pages 1–18. Springer US, 1995.

[AS07] James Aspnes and Gauri Shah. Skip Graphs. *ACM Trans. Algorithms*, 3(4), November 2007.

[AUS+09] K. Abe, T. Ueda, M. Shikano, H. Ishibashi, and T. Matsuura. Toward Fault-Tolerant P2P Systems: Constructing a Stable Virtual Peer from Multiple Unstable Peers. In *Proceedings of the 1st International Conference on Advances in P2P Systems (AP2PS'09)*, pages 104 –110, October 2009.

[BBC+11] Jason Baker, Chris Bondç, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the 2011 Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, NY, USA, 2005. ACM.

[BBP+11] W. Binder, D. Bonetta, C. Pautasso, A. Peternier, D. Milano, H. Schuldt, N. Stojnić, B. Faltings, and I. Trummer. Towards Self-Organizing Service-Oriented Architectures. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 115–121, July 2011.

[BCF+06] Walter Binder, Ion Constantinescu, Boi Faltings, Klaus Haller, and Can Türker. A Multiagent System for the Reliable Execution of Automatically

Composed Ad-hoc Processes. *Autonomous Agents and Multi-Agent Systems*, 12(2):219–237, 2006.

[BCPV04] Antonio Brogi, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing Web Service Choreographies. *Electron. Notes Theor. Comput. Sci.*, 105:73–94, December 2004.

[BD00] Thomas Bauer and Peter Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 94–109, London, UK, 2000. Springer-Verlag.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BG85] P. A. Bernstein and N. Goodman. Serializability Theory for Replicated Databases. *J. Comput. Syst. Sci.*, 31(3):355–374, December 1985.

[BGE99] Laszlo Boszormenyi, Herbert Groiss, and Robert Eisner. Adding Distribution to a Workflow Management System. In *10th International Workshop on Database and Expert Systems Applications (DEXA 99)*, pages 17–21, 1999.

[BGS+11] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1071–1080, NY, USA, 2011. ACM.

[Bla00] Uyless Black. *IP Routing Protocols: RIP, OSPF, BGP, PNNI and Cisco Routing Protocols*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[Bow00] F. D. J Bowden. A Brief Survey and Synthesis of the Roles of Time in Petri Nets. *Math. Comput. Model.*, 31(10-12):55–68, May 2000.

[BP09] Biörn Biörnstad and Cesare Pautasso. Let It Flow: Building Mashups with Data Processing Pipelines. In Elisabetta Di Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin Heidelberg, 2009.

[BP14] M. Babazadeh and C. Pautasso. The Stream Software Connector Design Space: Frameworks and Languages for Distributed Stream Processing. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 1–10, April 2014.

[BPA06] B. Biornstad, C. Pautasso, and G. Alonso. Control the Flow: How to Safely Compose Streaming Services into Business Processes. In *Services Computing, 2006. SCC '06. IEEE International Conference on*, pages 206–213, Sept 2006.

[Bre08] Gert Brettlecker. *Efficient and Reliable Data Stream Management*. PhD thesis, University of Basel, 2008.

[BS07] Gert Brettlecker and Heiko Schuldt. The OSIRIS-SE (Stream-enabled) Infrastructure for Reliable Data Stream Management on Mobile Devices. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1097–1099, NY, USA, 2007. ACM.

[BS11a] Gert Brettlecker and Heiko Schuldt. Reliable Distributed Data Stream Management in Mobile Environments. *Inf. Syst.*, 36(3):618–643, May 2011.

[BS11b] Gert Brettlecker and Heiko Schuldt. Reliable distributed data stream management in mobile environments. *Information Systems*, 36(3):618 – 643, 2011. Special Issue on {WISE} 2009 - Web Information Systems Engineering.

[BSS05] Gert Brettlecker, Heiko Schuldt, and Hans-Jürg Schek. Towards reliable data stream processing with OSIRIS-SE. In *Proceedings of the BTW Conference*, pages 405–414, 2005.

[CDCR02] Miguel Castro, Peter Druschel, Y. Charlie, and Hu Antony Rowstron. Exploiting Network proximity in Peer-To-Peer Overlay Networks. Technical report, Microsoft Research, 2002.

[CDE+12] J. C. Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[CGN97] Ting Cai, Peter A. Gloor, and Saurab Nog. DartFlow: A Workflow Management System On The Web Using Transportable Agents. Technical report, Dartmouth College, Hanover, NH, USA, 1997.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011.

[Cha04] D.A. Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's Highly Available Key-Value Store. In *Proceedings of the ACM Symposiumon Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[DK76] F. DeRemer and H.H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, 2(2):80–86, 1976.

[DLS+04] Frank Dabek, Jinyang Li, Emil Sit, et al. Designing a DHT for Low Latency and High Throughput. In *Proceedings of the 1st NSDI*, pages 85–98, 2004.

[Doo09] Kevin Dooley. *Designing Large Scale Lans*. O'Reilly Media Inc., Sebastopol, 2009.

[Edl15] Stefan Edlich. List Of NoSQL Databases. http://nosql-database.org/, 2015.

[FDKC06] Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer. Abstract Pushing Chord into the Underlay: Scalable Routing for Hybrid MANETs. Technical Report 2006-12, University of Karlsruhe, 2006.

[Fit04] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, August 2004.

[FJLM05] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES Distributed Publish/Subscribe System. In *Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05)*, pages 12–30, 2005.

[FY09] Sonia Gaied Fantar and Habib Youssef. Locality-Aware Chord over Mobile Ad Hoc Networks. In *Proceedings GIIS*, June 2009.

[GAH07] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *Proceedings of the 2005/2006 International Conference on Databases, Information Systems, and Peer-to-Peer Computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[Gao] Shengkui Gao. A Survey of Latest Performance, Development and Measurement Issues of Smart Phones Design, http://www.cse.wustl.edu/ jain/cse567-11/ftp/smartphn/index.html.

[GBK+11] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, et al. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[GHTC13] Katarina Grolinger, Wilson A. Higashino, Abhinav Tiwari, and Miriam A. M. Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22+, 2013.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[GL06] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006.

[GM04] Danny Goodman and Michael Morrison. *JavaScript bible (5. ed.).* Wiley, 2004.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[GYJ11] Siddarth Ganesan, Young Yoon, and Hans-Arno Jacobsen. NIñOS Take Five: The Management Infrastructure for Distributed Event-driven Workflows. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 195–206, NY, USA, 2011. ACM.

[Hab10] Mocky Habeeb. *A Developer's Guide to Amazon SimpleDB*. Addison-Wesley Professional, 1st edition, 2010.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 779–790, Washington, DC, USA, 2005. IEEE Computer Society.

[Hed88] C. L. Hedrick. Routing Information Protocol. RFC 1058, 1988.

[HGRW06] Tobias Heer, Stefan Gotz, Simon Rieche, and Klaus Wehrle. Adapting Distributed Hash Tables for Mobile Ad Hoc Networks. In *Proceedings PerCom 2006 Workshops*, Pisa, Italy, March 2006.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[HMA09] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed Similarity Search in High Dimensions Using Locality Sensitive Hashing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 744–755, NY, USA, 2009. ACM.

[HP08] Thomas Heinis and Cesare Pautasso. Automatic Configuration of an Autonomic Controller: An Experimental Study with Zero-Configuration Policies. In *2008 International Conference on Autonomic Computing, ICAC 2008, June 2-6, 2008, Chicago, Illinois, USA*, pages 67–76, 2008.

[HPA05] T. Heinis, C. Pautasso, and G. Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *Proceedings of the Second International Conference on Autonomic Computing. ICAC 2005.*, pages 27–38, June 2005.

[HXZ07] Jeong-Hyon Hwang, Ying Xing, and Stan Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proceedings of the ICDE*, 2007.

[Jen96] Kurt Jensen. *Coloured Petri Nets (2nd Ed.): Basic Concepts, Analysis Methods and Practical Use: Volume 1*. Springer-Verlag, London, UK, 1996.

[JKG09] S. Jafar, A. Krings, and T. Gautier. Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing. *IEEE Transactions on Dependable and Secure Computing*, 6(1):32 –44, January-March 2009.

[JL84] William B Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1–1, 1984.

[JM96] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, 1996.

[Jon87] James V. Jones. *Integrated Logistics Support Handbook*. TAB Books, Blue Ridge Summit, PA, USA, 1987.

[KBG08] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant Stream Processing Using a Distributed, Replicated File System. *Proceedings VLDB Endowment*, 1(1):574–585, August 2008.

[KBG+10] Martin Keen, Bryan Brown, Andy Garratt, Benjamin Käckenmeister, Ahmed Khairy, Kevin O'Mahony, and Lei Yu. *Building IBM Business Process Management Solutions Using WebSphere V7 and Business Space*. Number SG24-7861-00 in IBM Redbooks. IBM, May 2010.

[KCS10] R. Kaur, R.K. Challa, and R. Singh. Antecedence Graph based Checkpointing and Recovery for Mobile Agents. In *Proceedings of the IEEE International Conference on Communication Control and Computing Technologies (ICCCCT'2010)*, pages 419 –424, October 2010.

[KEAAH05] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi. A Statistical Theory of Chord Under Churn. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, pages 93–103, Berlin, Heidelberg, 2005. Springer-Verlag.

[KMW03] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-Oriented Composition in BPEL4WS. In *WWW (Alternate Paper Tracks)*, 2003.

[Lam] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March.

[Lam98] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam06] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[LM04] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a Decentralized Structured Storage System. *SIGOPS Operating System Reviews*, 44:35–40, April 2010.

[LMaJ10] Guoli Li, Vinod Muthusamy, and Hans arno Jacobsen. A distributed service oriented architecture for business process execution. *ACM TWEB*, 2010.

[LSPF07] Kevin Lee, Rizos Sakellariou, Norman W. Paton, and Alvaro A. A. Fernandes. Workflow Adaptation As an Autonomic Computing Problem. In *Proceedings of the 2nd Workshop on Workflows in Support of Large-scale Science*, WORKS '07, pages 29–34, NY, USA, 2007. ACM.

[LWW10] Che-Liang Liu, Chih-Yu Wang, and Hung-Yu Wei. Crosslayer Mobile Chord P2P protocol design for VANET. *Int. J. Ad Hoc Ubiquitous Comput.*, 6(3), August 2010.

[LYE+11] B. Loesgen, C. Young, J. Eliasen, S. Colestock, A. Kumar, and J. Flanders. *Microsoft BizTalk Server 2010 Unleashed*. Unleashed. Pearson Education, 2011.

[MB09] A. Mosincat and W. Binder. Automated performance maintenance for service compositions. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 131–140, Sept 2009.

[MB11] Adina Mosincat and Walter Binder. Automated maintenance of service compositions with SLA violation detection and dynamic binding. *International Journal on Software Tools for Technology Transfer*, 13(2):167–179, 2011.

[MJ07] Qi Meng and Hong Ji. MA-Chord: A New Approach for Mobile Ad Hoc Network with DHT Based Unicast Scheme. In *International Conference on Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007.*, pages 1533–1536, Sept 2007.

[MJB05] Alberto Montresor, Márk Jelasity, and Özalp Babaoglu. Chord on Demand. In *Proceedings of the fifth IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 87–94, Konstanz, Germany, August/September 2005. IEEE Computer Society.

[Moe12] Thorsten Moeller. *Flexible semantic service execution*. PhD thesis, University of Basel, 2012.

[MS10a] Diego Milano and Nenad Stojnić. Shepherd: Node Monitors for Fault-tolerant Distributed Process Execution in OSIRIS. In *Proceedings of the 5th International Workshop on Enhanced Web Service Technologies*, WEWST '10, pages 26–35, NY, USA, 2010. ACM.

[MS10b] T. Möller and H. Schuldt. OSIRIS Next: Flexible Semantic Failure Handling for Composite Web Service Execution. In *2010 IEEE Fourth International Conference on Semantic Computing (ICSC)*, pages 212–217, Sept 2010.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[MWSP08] Peter Merz, Steffen Wolf, Dennis Schwerdel, and Matthias Priebe. A Self-Organizing Super-Peer Overlay with a Chord Core for Desktop Grids. In *Proceedings of the 3rd International Workshop on Self-Organizing Systems (IWSOS)*, pages 23–34, Vienna, Austria, December 2008. Springer.

[NS73] I. Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Not.*, 8(8):12–26, August 1973.

[Oa04] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[Ora01] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[PA04] Cesare Pautasso and Gustavo Alonso. JOpera: a Toolkit for Efficient Visual Composition of Web Services. *International Journal of Electronic Commerce (IJEC)*, 9:107–141, Winter 2004/2005 2004.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings SIGCOMM*, pages 234–244, London, UK, 1994.

[PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, 2003.

[PDH04] H. Pucha, S.M. Das, and Y.C. Hu. Ekta: An Efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks. In *Proceedings WMCSA*, December 2004.

[PPBB14] Achille Peternier, Cesare Pautasso, Walter Binder, and Daniele Bonetta. High-performance execution of service compositions: a multicore-aware engine design. *Concurrency and Computation: Practice and Experience*, 26(1):71–97, 2014.

[PSH13] Lukas Probst, Nenad Stojnić, and Schuldt Heiko. COMPASS – Latency Optimal Routing in Heterogeneous Chord-based P2P Systems. Technical report, University of Basel, Switzerland, 2013.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[Rei13] Wolfgang Reisig. *Understanding Petri nets : modeling techniques, analysis methods, case studies*. Springer, Berlin, Heidelberg, 2013. DEBSZ.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, NY, USA, 2001. ACM.

[RGG+05] Sean Rhea, Brighten Godfrey, Karp Godfrey, et al. OpenDHT: a public DHT service and its uses. In *Proceedings of the 2005 Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, 2005.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[RST11] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a Scalable, Consistent, and Highly Available Datastore. *Proceedings of the VLDB Endowment*, 4:243–254, January 2011.

[SGF02] Rüdiger Schollmeier, Ingo Gruber, and Michael Finkenzeller. Routing in Mobile Ad Hoc and Peer-to-Peer Networks. A Comparison. In *Int'l Workshop on Peer-to-Peer Computing. In Networking 2002*, pages 172–186, May 2002.

[SH05] Munindar P. Singh and Michael N. Huhns. *Service-oriented computing - semantics, processes, agents.* Wiley, 2005.

[SH12] Nenad Stojnić and Schuldt Heiko. Safety Ring: Fault-tolerant Distributed Process Execution in OSIRIS. Technical report, University of Basel, Switzerland, 2012.

[SJV+15] Stefan Schulte, Christian Janiesch, Srikumar Venugopal, Ingo Weber, and Philipp Hoenisch. Elastic business process management: State of the art and open challenges for bpm in the cloud. *Future Generation Computer Systems*, 46(0):36 – 50, 2015.

[SKG+12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, NY, USA, 2001. ACM.

[SPS13] Nenad Stojnić, Lukas Probst, and Heiko Schuldt. COMPASS - Optimized Routing for Efficient Data Access in Mobile Chord-Based P2P Systems. In *Proceedings of the 2013 IEEE 14th International Conference on Mobile Data Management - Volume 01*, MDM '13, pages 46–55, Washington, DC, USA, 2013. IEEE Computer Society.

[SRHS10] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *Proceedings of the*

10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 448–454, Washington, DC, USA, 2010. IEEE Computer Society.

[SS12] N. Stojnić and H. Schuldt. OSIRIS-SR: A Safety Ring for self-healing distributed composite service execution. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 21–26, June 2012.

[SS13] Nenad Stojnić and Heiko Schuldt. OSIRIS-SR: A Scalable Yet Reliable Distributed Workflow Execution Engine. In *Proceedings of the 2nd ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '13, pages 3:1–3:12, NY, USA, 2013. ACM.

[SST+05] C. Schuler, H. Schuldt, C. Türker, R. Weber, and H.-J. Schek. Peer-to-peer Execution of (transactional) Processes. *International Journal of Cooperative Information Systems*, 14(4):377–406, 2005.

[STS+06] C. Schuler, C. Türker, H.-J. Schek, R. Weber, and H. Schuldt. Scalable Peer-to-Peer Process Management. *International Journal of Business Process Integration and Management (IJBPIM)*, 1(2):129–142, 2006.

[SWSjS04] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans j. Schek. Scalable peer-to-peer process management - The OSIRIS approach. In *Proceedings of the 2nd International Conference on Web Services (ICWS'2004*, pages 26–34. IEEE Computer Society, 2004.

[TC03] Liying Tang and Mark Crovella. Virtual Landmarks for the Internet. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, pages 143–152, NY, USA, 2003. ACM.

[VDAHW03] Wil Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. Business Process Management: A Survey. In *Proceedings of the 2003 International Conference on Business Process Management*, BPM'03, pages 1–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[VDAS11] Wil Van Der Aalst and Christian Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. The MIT Press, 2011.

[Vos09] Gottfried Vossen. *Transaction Models – the Read/Write Approach*. Springer US, 2009.

[WG08] Peter Y. Wong and Jeremy Gibbons. A Process Semantics for BPMN. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 355–374, Berlin, Heidelberg, 2008. Springer-Verlag.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[WZS04] R. Winter, T. Zahn, and J. Schiller. Random Landmarking in Mobile, Topology-Aware Peer-To-Peer Networks. In *Proceedings of the FTDCS*, Suzhou, China, May 2004.

[Ye06] Xinfeng Ye. Towards a Reliable Distributed Web Service Execution Engine. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 595 –602, September 2006.

[Yu10] Weihai Yu. Fault Handling and Recovery in Decentralized Services Orchestration. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 98–105, NY, USA, 2010. ACM.

[YY07] Weihai Yu and Jie Yang. Continuation-Passing Enactment of Distributed Recoverable Workflows. In *22nd Annual ACM Symposium on Applied Computing (SAC 2007), Seoul, Korea, March 11 - 15*, pages 475–481, 2007.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[Zel11] Florian Zeller. Programming in the Large based on OSIRIS-PRO . Master's thesis, University of Basel, 2011.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[ZS06] Thomas Zahn and Jochen Schiller. DHT-based Unicast for Mobile Ad Hoc Networks. In *Proceedings of the PerCom 2006 Workshops*, Pisa, Italy, March 2006.

[Zub80] W. M. Zuberek. Timed Petri Nets and Preliminary Performance Evaluation. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80, pages 88–96, NY, USA, 1980. ACM.

Curriculum Vitae

Nenad Stojnić

November 10, 1984	Born in Zagreb, Croatia Son of Brankica and Slobodan Stojnić Citizen of the Republic of Serbia and the Republic of Croatia
1990–1994	Primary School, Zagreb, Croatia
1994–1995	Primary School "Albert Schweizer", Osnabrück, Germany
1995–1998	Secondary School "Ernst Moritz Arndt Gymnasium", Osnabrück, Germany
1998–1999	Secondary School "First Vojvodian Brigade", Novi Sad, Serbia
1999–2003	High School of Electrical Engineering "Mihajlo Pupin", Novi Sad, Serbia
2003–2008	Master of Science in Computer Science University of Novi Sad, Faculty of Technical Sciences , Novi Sad, Serbia
2008–2010	Software Engineer with Schneider Electric, Novi Sad, Serbia
2010–2015	Research and teaching assistant in the group of Prof.Heiko Schuldt Database & Information Systems, University of Basel, Switzerland