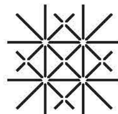
Exploring Online Evolution of Network Stacks

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

Pierre Imai
今井ピェア

aus Lörrach, Deutschland



Basel, 2013

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Christian Tschudin
Prof. Dr. Thomas Plagemann

Basel, den 12. November 2013

Prof. Dr. Jörg Schibler (Dekan)

Attribution-Noncommercial-No Derivative Works 2.5 Switzerland

You are free:

to Share — to copy, distribute and transmit the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license) available in German:
http://creativecommons.org/licenses/by-nc-nd/2.5/ch/legalcode.de

Disclaimer:
The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) — it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license. Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Quelle: http://creativecommons.org/licenses/by-nc-nd/2.5/ch/deed.en Datum: 3.4.2009

Abstract

Network stacks today follow a one-size-fits-all philosophy. They are mostly kept unmodified due to often prohibitive costs of engineering, deploying and administrating customisation of the networking software, with the Internet stack architecture still largely being based on designs and assumptions made for the ARPANET 40 years ago. We venture that heterogeneous and rapidly changing networks of the future require, in order to be successful, run-time self-adaptation mechanisms at different time scales and based on continuous performance measurements.

For this purpose we present an autonomous stack composition framework and configuration logic inspired by biological evolution: Stack configurations (compositions) compete against each other on-line, new compositions evolve from the best previous performers. Compositions are further selected and pooled together according to the traffic and network conditions, forming a long-term situation-aware knowledge base.

We demonstrate the feasibility of our runtime adaptive approach by exposing our implementation to simulated as well as real world Internet traffic. Beyond individual "zero knob protocols" we show that our network management system not only tunes a network stack's parameters but can also change its composition on the fly.

This lowers the barrier for introducing novel protocols, move to other run-time systems or accommodate new traffic patterns. Ultimately, this lets engineers of future computer networks focus on specialised rather than smallest common denominator solutions, as the run-time choice and management is taken care of by our system.

Acknowledgements

I would like to thank all the people who helped and supported me in the process of writing this thesis and during my research.

First of all, I wish to thank Christian Tschudin for the opportunity to freely research a topic of my own choosing, a freedom that is very rare and which I am very glad to have experienced here.

I would like to express my sincere gratitude to Thomas Plagemann for agreeing to examine and judge this thesis.

Furthermore I would like to thank Thomas Meyer and Manolis Sifalakis for the frequent discussions throughout my research and their critique of my sometimes incomprehensible utterances which later somehow evolved into this – now hopefully readable – document.

I wish to also thank all the people who helped me by either proof-reading this document – Massimo Monti, Ghazi Bouabene – or by collaborating with me for their B.Sc. thesis work – Yannic Kilcher and Florian Lindörfer.

Most importantly, I would like to thank my wife, Shihori, for all the nights and days she spent by my side while I was working on this document, and for all the tea she supplied me with to keep me awake.

And finally, I wish to thank you, the reader, for actually reading (at least parts of) this thesis.

Contents

1 Introduction & Motivation **1**

1.1 A Brief History of the Internet 2

1.2 A Vision of the Future (Internet) 3

1.3 The Expectation 4

1.4 Contributions & Thesis Layout 5

2 State of the Art & Technologies **9**

2.1 Autonomy Research Fields 9

2.2 Network Autonomy on the Conceptional Level 16

2.2.1 Attributes of Autonomy 16

2.2.2 Awareness and Reason in the Network – The Knowledge Plane 17

2.3 Implementing Autonomy – The Autonomic Control Loop . . 19

2.4 Collection of Information 20

2.4.1 Measuring Methods & Metrics 20

2.4.2 Cross-Layer Designs 23

2.5 Analysis of Information 23

2.5.1 On-line Experimentation 24

2.5.2 Goal- & Utility-Measures 26

2.5.3 Classification & Detection 27

2.6 Decision 29

2.6.1 Evolutionary Algorithms 30

2.6.2 Reinforcement Learning 31

2.6.3 Swarm Intelligence 32

2.6.4 Other Approaches Inspired by Nature 33

2.6.5 How to Choose and Configure an Optimization Algorithm . . . 34
2.7 Action . . . 34
2.7.1 Protocols & Stacks . . . 35
2.7.2 Dynamic Resolution of Names & Functionality . . . 40
2.7.3 Run-time Code Deployment . . . 42
2.8 Summary & Conclusion . . . 43

3 Rationale and Architecture **45**
3.1 Towards Autonomous Stack Evolution . . . 45
3.1.1 Continuous Optimization . . . 46
3.1.2 Situational Awareness . . . 46
3.1.3 Flexible Goals and Environments . . . 46
3.1.4 Gradual and Local Adaptation . . . 47
3.1.5 Situational Memory . . . 47
3.1.6 Distributed Multi-node Optimization . . . 47
3.2 Concepts & Features . . . 48
3.3 Stack Composition System . . . 51
3.3.1 Layered Evolution – Long-term Stack Evolution vs. Mid-/Short-Term Adaptation . . . 52
3.3.2 Framework Layout . . . 54
3.4 Stack Layout & Specifications . . . 54
3.4.1 Stack Modules . . . 57
3.4.2 Stack Composition . . . 59
3.5 Stack Mechanics . . . 62
3.5.1 Stack Steering System . . . 62
3.5.2 Stack Composer . . . 63
3.5.3 Persistent Storage . . . 63
3.6 Evolution Machinery . . . 64
3.7 Handling Modular Stacks . . . 66
3.7.1 Module Instance Life-cycle . . . 66
3.7.2 Interchangeable Modules . . . 67
3.7.3 Unified Communication Interface . . . 70
3.7.4 Reliable Communication Channel . . . 74
3.7.5 Sender-Initiator-Defined Composition . . . 75
3.7.6 In-stack Redirectors . . . 77
3.8 Chapter Summary . . . 79

4 Evolution Logic **81**
4.1 Long-Term Decisions – Evolution Engine . . . 83
4.1.1 Configuration Space & Fitness Landscape . . . 83

4.1.2 Calling Conventions 86
4.1.3 Algorithms for Controlled Evolution 87
4.1.4 Selecting the Evolution Logic 109
4.2 Mid-Term Decisions – Classification & Population Selection 112
4.2.1 Matrix-based Selection 113
4.2.2 k-Means Clustering Adapted for Population Selection 114
4.2.3 Choosing Classification Criteria 118
4.3 Short-Term Decisions & Protocol Multiplexing 119
4.3.1 In-stack Redirectors 119
4.3.2 Decision Modules 121
4.4 Summary & Conclusion 121

5 Stabilising the Measurement Environment 123
5.1 Sensor Information 124
5.1.1 Passive Sensors 124
5.1.2 Active Sensors 124
5.1.3 Sensor Requirements 125
5.2 The Problem of Noise 125
5.3 Stabilising the Measurement Environment 131
5.3.1 Measurements & Data Processing 131
5.3.2 Experimentation 132
5.3.3 Ensuring Comparability 137
5.3.4 Information Exchange 138
5.3.5 Fitness Calculation 140
5.4 Chapter Summary 144

6 Experimental Validation 145
6.1 Evaluation of the Long-Term Stack Evolution Approach . . . 146
6.1.1 Scenario 1A: Composition of the Internet Protocol Stack – Simulation 147
6.1.2 Scenario 1B: Composition of the Internet Protocol Stack – Physical Test-Bed 155
6.1.3 Scenario 2: Error Correction & Compression 157
6.1.4 Scenario 3: Bias Towards Specific Configurations . . 165
6.1.5 Summary and Conclusions regarding Long-Term Stack Evolution 172
6.2 Evaluation of the Situational Classification Approach 174
6.2.1 Detecting Changes in the Network Conditions . . . 174
6.3 General Conclusions Derived from the Experiments 183
6.3.1 Long-Term Stack Evolution 183
6.3.2 Mid-Term Adaptation 183

6.3.3 Applicability 184

7 Discussion 185

7.1 Thesis Summary 185

7.2 Contributions, Limitations & Future Work 187

7.2.1 Architecture 187

7.2.2 Cognitive & Learning Facilities 189

7.2.3 Situational Awareness & Knowledge Base 192

7.2.4 Information Gathering & Assessment 193

7.3 Concluding Remarks 193

A Protocol Design Requirements 195

A.1 Monolithic and Flexible Protocol Design 196

A.2 Configurability 197

A.3 Reliability 197

A.4 Altruism 198

A.5 Co-operation 198

A.6 Restartable Protocols 199

A.7 Modularity 200

B Implementation and Experimentation Environment 201

B.1 Implementation 201

B.1.1 Stand-alone User-space Implementation 201

B.1.2 Simulation 203

B.1.3 Realistic Physical Environment 204

B.2 Implemented Protocol and Service Facilities 204

B.2.1 Internet Protocol Implementations 205

B.2.2 Other Protocols 207

Glossary 211

Bibliography 219

Chapter 1

Introduction & Motivation

The Internet stack constitutes the mainstay of most of the current communication infrastructure, as the World Wide Web, mobile telephone networks, as well as most networked appliances operate on top of it. But its inflexible layered and statically composed structure is often considered one of the main impediments for the deployment of more advanced technologies.[40] Replacement architectures may provide run-time, on-demand protocol and service assembly out of small building blocks, and adapt specifically to the requirements of the user and network. Such approaches promise performance improvements and could cater to environments where deploying the Internet stack is inadequate: Sensor networks, grid computing, ambient networks, etc., become ever more prevalent, but have widely divergent needs and require specialised stacks.[96] The necessary fine-tuning and optimisation to the application scenario is however very labour intensive. The interactions and influence between networking entities and traffic, the user demands, as well as their actions, are hard to know in advance, and often change over time – and sometimes rapidly and frequently so. Thus the optimal composition of stack modules and their configuration is hard or even impossible to derive off-line or through simulation. As the processing power and capabilities of standard networking hardware grows, we propose that such entities should find the optimal composition and configuration of the stack themselves. Whereas architectures and mechanisms for this purpose have been researched very extensively, the actual logic that realises this facility has so far been somewhat neglected. This thesis presents a first step towards a methodology and framework for this purpose, within which the network stack shall evolve over time by means of machine learning and on-line trial-and-error experimentation.

In this chapter we motivate our work by means of a honest yet unflattering description of why we perceive the Internet to be in need of a radical change. We introduce the problem space and outline the vision that inspired our work. We detail the expectations we intend to fulfil and specify our contributions. Lastly, we describe the layout of the thesis.

1.1 A Brief History of the Internet – And why it is so difficult to replace

The Internet stack has become predominant over the last thirty years, and now penetrates many application areas which until a few years ago were served by specialised networking protocols. Even telephony is now almost completely handled by All-IP packet-switched networks, with 4th-generation telephony networks being wholly based on IPv6.[2]

But contrary to what on first sight appears as a "success story", the Internet might soon lose one of its key features and become unable to host new and unanticipated applications.[37] The Internet's design made perfect sense for the client-server-oriented networks of the 1970s and 1980s,[61] but these networks have fairly little in common with the current or future Internet infrastructure.[64] The demands on the networks, the application spectra, and even the user base do not only diversify,[20,108,278,339] they cannot even be expected to remain static for a prolonged period of time.[66] The apparent need for a new architecture to replace the Internet is reflected by the sheer number of projects[3,7,16,36,56,67,147,246,257,258,384] that investigated alternative designs.[38,62,72,96,153,172,199,231,298,354] Yet even the rather conservatively designed IPv6, which can be gradually and non-disruptively deployed on the current infrastructure, is very slow on the uptake outside telephony networks.[345]

The hourglass[1] principle of the Internet, which intends the bottom and top of the stack to evolve and the centre to remain static, might very well have led to the Internet's success.[149] Innovative technologies, such as Wi-Fi, LTE, WiMAX, etc., flourish below the common substrate of IP which enables local and gradual deployment. New services sprout on top of HTTP,[285] because the necessary investments are limited to the local infrastructure. But we suspect that the same economical reasons prevented the Internet's further evolution. Changes to the Internet backbone itself became prohibitively expensive, as the homogeneous waist would have to be replaced more-or-less as a whole. The view that physical replacement of the ossified Internet stack is not

[1] The waist of the hourglass is constituted by the rigid enforcement of the IP protocol(s), but it has been argued that HTTP nowadays represents the new waist.[285]

viable might explain the current surge of efforts for virtualisation.[18] After all, technologies such as peer-to-peer and other overlay networks, VLAN, VPN, OpenFlow,[235] GENI,[135] PlanetLab,[283] cloud computing, etc., help to avoid the costs that physical deployment would induce. But often it is not so much the cost of the hardware itself that prevents change. Instead the salaries of the engineers and administrators that set-up, configure, and maintain the network have the highest impact.[191]

1.2 A Vision of the Future (Internet)

For the future we envision network machinery that freely and autonomously configures and composes the network stack out of atomic and modular protocol building blocks.

While several replacement architectures intend the imitate the thin waist of the Internet,[257,384] we venture that a monolithic transport is not sufficient for future. We further claim that we should not impose layering as it promotes redundancy.[355] Instead the architecture has to be flexible enough to handle arbitrary new protocols, services, and combinations or configurations thereof. This notion is supported by many other recent proposals, which intend to replace the layered stack with silos,[96] heaps,[40] compartments,[38] services,[379] recursive designs,[354] or even multiple parallel architectures.[367]

We reason that only an architecture or network stack specifically tailored to the current requirements of the applications and users can offer the best possible performance, in the same way as specialised protocols are often superior to generic ones.[109] Research into cross-layer adaptation seems to further supports this claim.[54,98,136,193] Likewise, specialised stacks, tailored for a specific application area, e.g. those developed for low-power sensor networks[216] or the high-performance requirements of e-Science applications,[158] offer even bigger advantages over generic stacks, as they do not have to compromise to the needs of different environments.

Instead of crafting specialised stacks by hand – which is a very labour intensive task – we expect that the systems of the future will autonomously adapt and optimise the stack for their own needs. More than ten years ago, IBM stated that future software and systems need to be able to manage and configure themselves, because the workload would otherwise become almost unmanageable and unaffordable.[191]

We assume that the future Internet is in even more dire need of such autonomous adaptation: The optimal behaviour of the network depends on the traffic, applications, users, etc. And even the notion of optimality is not

static, as the expectations of the users also change over time.[66] Simulating the current Internet is a very difficult task,[15,118,276] and the networks of the future will be even larger, more flexible, and more heterogeneous. We thus claim that the entities in the network need not only to sense, decide, and adapt based on a policy or a simple control loop, but have to be context-aware[133,262] and capable of explorative[296,357] learning.[21,63,244,292] We envision that the entities in the networks of the future will autonomously modify their network stacks, measure how they perform within the physical network environment by exposing them to the actual traffic therein, and learn how to adapt based on the measured effects of these actions, in the same fashion as autonomous robots already do.[42] Feedback for this purpose could in some cases be derived by means of simple calculations from network measurements.[189,368] But we expect the dynamics of the networks and in particular the intentions of the users to be impossible to predict at the time of design or deployment. Just as ADSL was designed for consumers of downloadable content, which do not need much upstream capacity, but was primarily used for synchronous peer-to-peer traffic, the future of computer networks is unpredictable. Thus the feedback should ideally be based on the perceived utility for the users.[211,212]

1.3 The Expectation

While we would certainly like to be able to realise our vision of autonomous on-line network stack evolution, it is still far from reality, and much research remains to be done. We therefore concentrated on the development of a system capable of autonomously improving the network stack's composition and configuration based on an arbitrary utility measure. For this purpose the system shall experiment by trial-and-error, i.e. create new stack configurations on its own, expose them to the actual network traffic, measure the effect by means of sensory input gathered from the network, and then cognitively decide how to further improve the configuration and composition through machine learning methods. This adaptation process shall solely be based on feedback in the form of a user-defined utility measure, the fitness function, i.e. the system should not possess any intrinsic knowledge of the protocol behaviour or of what constitutes a good solution, as both notions may change at run-time. Due to the non-deterministic nature of this process, we do not expect the system to arrive at the optimal solution. Instead we intend to investigate with what probability and how fast the system can arrive at a close-to-optimal solution for a limited, but specific set of test cases. Autonomous adaptation is complicated by the unpredictable nature of the

network and traffic conditions on which the utility of a specific stack configuration depends. The system thus has to be context-aware, i.e. able to evaluate the current situation of the network, the traffic and itself, and based on this evaluation to decide how to modify the stack. Furthermore it has to be able to ensure that the results of experiments are comparable, i.e. that the utility of stacks is evaluated under sufficiently similar conditions as not to affect the perceived utility.

1.4 Contributions & Thesis Layout

Our aim is to design a system for autonomous on-line network stack evolution, i.e. a system which offers the advantages of a specialised network stack, which is adapted to the present conditions of the network, suitable for the current traffic therein, and which offers near-optimal utility as defined by its users. For this purpose, we envision an autonomous stack composition system, which can monitor the state of the network, the ongoing traffic, gather user feedback, and create a stack that capable of adequately fulfilling the user expectations. To realise this vision, we researched the feasibility of

1. an architecture to support autonomous stack composition,
2. logic which controls the stack's long-term evolution,
3. logic to realise situational awareness and short- to mid-term adaptation,
4. and mechanisms for the reliable gathering and assessment of (measurement) data.

Overall, our work should be seen as a proof-of-concept which shows that our design makes autonomous evolution of network stacks and adaptation to the environment feasible. We support this claim by selected experiments, and consider a more thorough exploration of the problem space for future work.

Our first contribution is the realisation of (a subset of) Clark *et al.*'s concept of a knowledge plane[63] for communication by means of the classic feedback loop first proposed by Wiener.[376] This includes the architecture – introduced in *Chapter 3* – and implementation of a comprehensive framework for autonomous stack composition and adaptation based on experimentation and on-line measurements, which we use as the basis on which we

validate our research and as a test-bed for further exploration. Our framework replaces the operating system network stack with a protocol "composite"[3] which is dynamically configured and composed out of those micro-protocols[267] available at run-time. As a side-effect of implementing several micro-protocols, which encompass e.g. the functionality of the Internet stack, we derived requirements and guidelines for protocol design that enables effective stack composition. Additionally, we provide a means for placing short-term, e.g. per-flow or per-packet, adaptation facilities within the stack, as described in *Section 4.3.*

Our second contribution concerns the cognitive functionality that enables network stack evolution towards a goal state defined by an utility or fitness measure[73,97,189]. For this purpose we explored several common machine learning techniques[26,137,209,343] and – based on experimentally-gained knowledge – invented a new search algorithm specifically for the purpose of generating optimised candidate stack configurations, the Composition Tree Search. We describe these algorithms in detail in *Section 4.1.*

Our third contribution pertains to the situation-aware classification and adaptation logic, which we detail in *Section 4.2.* We investigated problems concerning the accurate measurement and assessment of the network and traffic conditions, and how to stack can be autonomously optimised even when the network or traffic conditions are unstable. We designed and implemented functionality that enables the framework to notice when the situation in the network changes and to select a more adequate stack based on an on-line classification process. The experimentation facilities also utilise these techniques, enabling the system to evolve several independent stack configurations suitable for different situations and to guarantee experimentation under sufficiently similar and thus comparable conditions. We thus realise a situational knowledge base for adaptation.

Our fourth contribution relates to the facilities for autonomous gathering of measurement data and assessment of the information therein, which we introduce in *Chapter 5.* We designed and implemented methodologies that enable the framework to explore the utility of candidate network stacks by means of trial-and-error experimentation. The framework decides when to schedule experiments based on the perceived utility of the current stack and the measured network and traffic conditions. These experiments are

[3] We denomimate this entity as "stack" throughout this document, but other forms of composites such as silos or heaps are also supported.

performed on-line, i.e. the stack is replaced by a new one created from a candidate stack configuration and exposed to the actual network traffic. We investigated if such experimentation at runtime is possible in the presence of unrelated traffic and experiments performed by other nodes. We devised and implemented measures to safeguard that the fitness assessment is not impeded by measurement inaccuracies or operational difficulties.

In this chapter we introduced some of the problems the Internet faces, as well as our vision for the networks of the future. We described how our research contributes to this vision. In the next chapter we introduce research and technologies related to our work. Afterwards we detail our design, beginning with the architecture of the stack composition system, followed by a discussion of the logic that guides the stack evolution process, then we discuss the problem of reliably gathering and assessing the information needed for this process. We then introduce some of the experiments we performed to validate and evaluate the system and conclude with a discussion of our work, the achievements and an outlook on future work.

Chapter 2

State of the Art & Technologies

As already stated in the previous chapter, we are not the first to notice the problems affecting the current Internet nor to research and propose new ideas and technologies for its improvement or even replacement. And as many before us, we assume that autonomy, in particular the ability of the network to rationally decide and adapt to the needs of the users will play a vital role in the future Internet.

In this chapter we therefore detail current and past research efforts and the state of the art in communication networks and autonomy research in general. We selected a representative subset of the research which we consider most important for our work and describe in detail how it influenced our work. We begin with a short introduction of the related research fields, and then present our perspective of autonomy, focussing on the aspects of networking we address in this thesis. Afterwards we detail research pertaining to the individual aspects of autonomy in the network, and conclude with a summary of the research discussed in this chapter.

2.1 Autonomy Research Fields

Research into autonomy in the context of computer networks was strongly influenced by Clark's proposal for a knowledge plane for the Internet,[63], IBM's manifesto which defines Autonomic Computing,[124] and Fraunhofer's research agenda for Autonomic Communication,[319] as we describe in more detail below. While the differences in terminology sometimes suggest otherwise, the fundamental concepts and definitions of autonomy used in the various areas of computer science are almost identical. We introduce some

Figure 2.1: *The Autonomic Computing Tree from* Towards an Autonomic Computing Environment[331].

of the related fields of research here and state why we consider them relevant for our own work. The distinction between these areas is often not very clear, as the solution domains overlap, and the only fundamental distinction is the definition of the problem scope. For example, some research in the field of Autonomic Communications could just as well be attributed to Autonomic Computing, or vice-versa, e.g. Saffre's method for autonomic service distribution,[309] or Hadjiantonis *et al.*[146] policy-based approach of context-awareness for the self-organization of ad-hoc networks. Likewise Demestichas *et al.*'s[77] M@ANGEL platform, a distributed framework for managing future mobile communication networks in which the individual elements of which the network is composed are called cognitive elements, can be attributed to Cognitive Networks as Fortuna *et al.*[121] do, but the authors themselves consider it part of Autonomic Computing.

Autonomic Computing

In 2001 IBM released a manifesto stating the need for autonomic computing due to the looming software complexity crisis, and backed up this claim by quantifying the costs of the installation and maintenance of current applications. Whereas the original document is no longer available on the Internet, a follow-up publication by Ganek[124] elaborates further on this claim, stating that e.g. the administrative costs can account for up to 75% of the cost of database ownership, and that *"Today's systems must evolve*

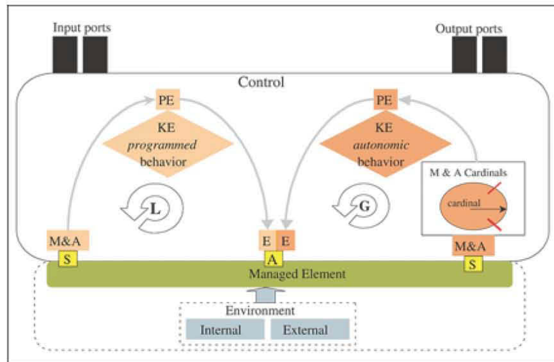


Figure 2.2: *An element of the autonomic computing environment, from* Autonomic computing: An overview[269].

to become much more self-managing, that is: self-configuring, self-healing, self-optimising, and self-protecting". *Figure 2.1* illustrates general properties that define autonomic computing. IBM's initiative formalised the problem and gave a new impetus to research in this field, but autonomic facilities were deployed in computer systems already some time before. For example, Kephart[190] developed an artificial immune system that autonomously located unknown virus variants and informed neighbouring nodes about the infection, and similar approaches have been used for intrusion detection.[154] Generally speaking, autonomic computing requires software and system architectures to be self-adaptive in all areas pertaining to its functionality, such as configurability, fault tolerance, maintainability, performance, security, etc. As Parashar *et al.*[269] elaborate, "[...]*in the case of emerging systems and applications, the specific requirements, objectives and choice of specific solutions (algorithms, behaviors, interactions, etc.) depend on runtime state, context, and content, and are not known a priori. The goal of autonomic computing is to use appropriate solutions based on current state/context/content and on specified policies.*" An autonomic system consists of self-contained autonomous elements, which interact with and react to their environment as shown in *Figure 2.2*. Its operation is controlled by a manager which consists of two control loops. The local loop reacts to changes in the local environment based on embedded knowledge. The global loop in turn is aware of the global environmental state, and may contain machine learning functionality to tackle unknown states of the environment or may depend on human interaction.

Several projects investigated the possibilities of autonomic computing. For example, AutoMate[5] focused on context-aware grid applications, trying to make them self-configuring, self-composing, self-optimising and self-adapting. It provides facilities for autonomic composition, adaptation, and optimisation by means of a decentralised deductive engine called RUDDER, which uses user-defined rules to supervise specialised agents distributed throughout the system and employs learning techniques for dynamic self configuration and rule adaptation. The Kinesthetics eXtreme[179] project tried to retrofit existing systems with autonomic properties. For this purpose it deployed sensors called gauges throughout the legacy system which aggregate and analyse the system state, based on which an autonomic control layer triggers actions such as reconfiguration tasks. The OceanStore[204] project provides global-scale persistent storage and utilises introspection to manage replica management and to recognise clusters of closely-related files, routing, recognition of unreliable peer, etc., and adapts autonomously to optimise its performance.

Autonomic Communications

The concepts of Autonomic Communications developed around 2003 based on an research agenda proposed by Smirnov.[319] According to Dobson *et al.*,[84] it *"seek[s] to improve the ability of network and services to cope with unpredicted change, including changes in topology, load, task, the physical and logical characteristics of the networks that can be accessed"*. The direction of research was strongly influenced by Tschudin's vision of large autonomous networks able to assemble themselves from zero state,[16] and Crowcroft's idea of opportunistic networks which asynchronously communicate using any available technology.[147] Analogous to Autonomic Computing introduced above, Autonomic Communications utilises self-management, self-configuration, and self-regulation to reduce the necessity of management efforts and, in particular, of manual intervention into network operations. As the heterogeneity and complexity of networks grows, furthered by the trend towards mobility, ubiquity, and pervasiveness of communications, the management overhead is feared to become almost unmanageable. Autonomy within the network shall help to thwart this threat by making the network able to adapt itself to the user requirements as specified e.g. through policy definitions, ontologies, etc.

Whereas the similarities between Autonomic Communications and Autonomic Computing are rather obvious a distinction is nevertheless justified. Autonomic Computing focuses mostly on application software and resource management aspects, but Autonomic Communications' vision goes

further. As Dobson[84] notes it encompasses a *"deep foundational rethinking of communication"* and a network whose components *"work in a totally unsupervised manner, able to self-configure, self-monitor, self-adapt, and self-heal – the so-called self-∗ properties"*. This vision is guided by Clark *et al.*'s proposal for a *knowledge plane*[63] in the Internet, which bridges the gap between the lower levels in the system, i.e. the network stack, etc., and the user- and application-level context-awareness. Furthermore, the network itself can provide context and semantic information. Due to the strong impact of Clark's proposal and its close relation to our own research we discuss it in detail in *Section 2.2.2* and also present related research projects there.

Cognitive Radio

The problems encountered by current radio technology have lead to research into how cognitive abilities could help improve e.g. the spectrum selection process. The concept of Cognitive Radio[157,245] utilises a control loop which is very similar to the approach discussed above, called the cognitive cycle: As core functionality, the radio device constantly monitors the available spectrum bands and detects holes therein. It then analyses the characteristics of these holes, determines data rate, transmission mode, and the bandwidth of the transmission, and finally decides on the appropriate spectrum band according to user requirements and the spectrum characteristics. The device is expected to cognitively choose the most appropriate operating frequency, modulation, transmission power, and communication technology. Mitola's vision[244] for future cognitive radio networks includes even far more advanced features, such as planning capabilities, video and voice analysis, location- and self-awareness, and the creation of sentient spaces.

Most of the current network-related research into Cognitive Radio, however, focuses on cross-layer and distributed optimisation tasks, as it is widely assumed that co-ordinated behaviour across layers or across nodes can lead to significant performance increases. For example, Zheng *et al.*[372] analyse how decoupled designs for routing and spectrum selection and collaborative architectures behave and come to the conclusion that cross-layer designs offer superior performance and in related research Zhao *et al.*[391] presented a distributed approach offering a performance increase of around 30% and a delay reduction of around 50%.

The most prominent application area for cognitive radio, as described by Devroye *et al.*,[79] is the use of licensed frequency bands by secondary, i.e. unlicensed, users in such a way that the operation by the primary user, e.g. a TV broadcast service, is not impeded, yet their own performance is maximised. Interest in this research is of strong commercial interest, as fre-

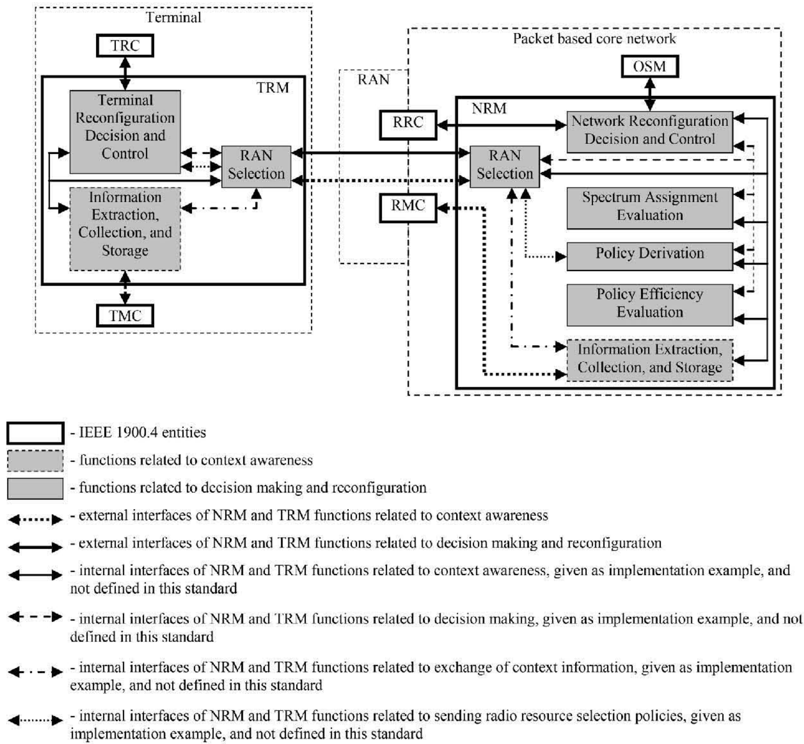


Figure 2.3: *The components of IEEE 1900-compatible radio equipment specify support for autonomous cognitive capabilities.* [166]

quency bands are scarce and the sharing of licensed bands seems promising: Jeon *et al.* [176] show that the presence of a secondary network does not significantly reduce the performance of the primary network, if appropriate routing methods are used. Similarly, Akyildiz *et al.* [8] proof that secondary networks can achieve the same optimal throughput as primary networks, and Thomas *et al.* [351] convincingly describe that the ability to fully cognitively control all nodes' radio operations significantly improves performance compared to partial control. Of particular interest in this respect is the upcoming IEEE standard 1900, [139,166,167] which defines e.g. the requirements for policy-based reasoning capabilities and explicitly includes support for context-aware autonomous self-reconfiguration, as shown in *Figure 2.3*.

Cognitive Networks

Some researchers propose the name Cognitive Networks for a field which is very similar to Autonomic Communications,[295] but the same term is however often used to refer to the Cognitive Radio Networks.[79] One definition of a Cognitive Network, which is closely related to Clark's concept of the knowledge plane, is given by Thomas *et al.*[350]: *"A cognitive network has a cognitive process that can perceive current network conditions, and then plan, decide and act on those conditions. The network can learn from these adaptations and use them to make future decisions, all while taking into account end-to-end goals."* Research in this field tries to extend the scope of reasoning from the protocol to the whole network, it encompasses a wide range of topics such as cross-layer design and distributed scheduling, etc., of which Fortuna *et al.*[121] provide an overview.

Active Networks

According to Tennenhouse and Wetherall's definition,[348] *"Active networks allow their users to inject customised programs into the nodes of the network"*, i.e. they contain switches which *"perform customised computations on the messages flowing through them"*,[347] and thus extend the possibilities for processing beyond the simple operations of classical networking, which were limited to the protocol headers. Active Networks are based on the principle that networks should be programmable like computers and extensible at run-time. Of particular interest for us is the capability of run-time extensibility, e.g. through mobile code deployment. In the mid-1990s, two approaches for active code deployment were envisioned, programmable switches which offer mechanisms for placing and executing programs on them and capsules which contain code in addition to the normal user payload of classical packet networks. We briefly discuss code deployment in *Section 2.7.3*. Programmable switch architectures include Click[197] and Router Plugins,[74] which we introduce in more detail in *Section 2.7.1*.

Autonomy in Software Engineering

Autonomy also plays an important role in current software engineering research. One of the goals of the Unity[52] project, for example, is to explore a technique called *goal-driven self-assembly*, in which the autonomic components of the system are assigned goals which they try to fulfil according to role-descriptions in a shared policy storage, and for which purpose they use other components according to their service descriptions. While the

history of dynamic re-composition of software reaches back to the beginnings of computing, when self-modifying code was a necessity due to lack of memory capacity and for runtime code optimisation, the interest in adaptive computer systems was rekindled by the emergency of ubiquitous and autonomous computing. McKinley *et al.*'s[236] article serves as an excellent introduction to these topics from a Software Engineering point-of-view.

For our purposes, the following research has particular relevance. Dellarocas *et al.*[76] propose an architecture for the field of Software Engineering, which has some significance for our research. Their *self-evolving software systems* autonomously adapt to changes in external conditions by dynamically reconfiguring the internal system and swapping software modules. They further introduce an evolution engine which monitors the execution of a running application and decides when and how to evolve it. Alagar *et al.*[10] present a framework for the construction of self-evolving real-time reactive systems. Their design consists of a Reuse Repository, which stores information on different versions of evolvable system components and their relations, an Evolution Engine, which calculates the reliability of the modules and the system constructed from them based on a Markov model, an Evolution Monitor which observes the environment and reacts to changes by adapting the system architecture based on the output of the Evolution Engine. Their design is based on formal models of the environment, software unit, and the desired properties of the system. Self-evolution of program code or protocols has also been been studied in other fields, as we discuss in *Section 2.6.4*.

2.2 Network Autonomy on the Conceptional Level

We decided to discuss the related work based on the concepts and terminology most commonly used in the context of Autonomous Communications,[84] as its vision most closely resemble our own.

2.2.1 Attributes of Autonomy

Autonomy is often defined by means of the following aspects, which are collectively referred to as "Self-∗".[21] Kephart and Chess describe[191] *self-configuration* as the autonomic configuration of systems and components based on high-level policies, and define *self-optimisation* as the continual exploration and learning of ways for improving the system operations: "[. . .]*autonomic systems will monitor, experiment with, and tune their own parameters and will learn to make appropriate choices about keeping*

functions or outsourcing them." *Self-verification* and *self-checking* safeguard against operational errors, *self-healing* is the ability to recognise and correct faults, and *self-protection* helps to prevent the occurrence of problems before they happen.

2.2.2 Awareness and Reason in the Network – The Knowledge Plane

The concept of autonomous networks, i.e. networks that – to some extent – are situation-aware and able to reason, is often associated with Clark *et al.*'s famous proposal for a knowledge plane for the Internet,[63] which therefore warrants a more elaborate discussion. Clark envisions an autonomous "meta-entity", the knowledge plane, which is located within the network and capable of cognitive reasoning, and as a pre-requisite context-aware. They consider the following attributes vital its realisation. The first attribute is *edge involvement*, i.e. the sharing of information beyond the traditional boundaries of the network, and thus involves e.g. user-preference- or goal-related knowledge. As second attribute they propose a *global perspective*, in other words local knowledge, for example information gathered only within the local subnet, is insufficient to guarantee the effective operation of the knowledge plane, which should instead extend its perspective to include knowledge and observations gathered in other parts of the network. Thirdly they assume the knowledge plane's structure be *composable* of modular elements, e.g. sub-planes, so that two independent networks should be able to collaborate and exchange knowledge and services, yet has to be able to cope with the heterogeneity of these components, which might for various reasons be reluctant or incapable to share all available knowledge. They further demand an *unified* approach, as to prevent the partitioning of knowledge which lead, for example, to the problems that motivated research into cross-layer architectures. Lastly, and maybe most importantly, they demand *cognitive* capabilities, as the knowledge plane needs to be able to operate in an environment about which it possesses incomplete and sometimes even incorrect information, and yet has be able to reason about how to most efficiently fulfil its – again possibly underspecified – goals. Apart from the control loop we introduce below, they consider the ability to *learn* vital for the knowledge plane's ability to achieve these objectives, and even suggest several possibilities to how this capability might be achieved, e.g. *"by trial and error"*.

The cognitive capabilities demanded by Clark *et al.*'s knowledge plane arguably pose the greatest challenge to its realisation. Luckily, research pertaining to other fields of (computer) science, in particular artificial intelligence concern themselves with similar topics and for an already consider-

ably longer timespan, and thus naturally strongly influence the design decisions and learning techniques proposed for the knowledge plane. Clark *et al.*themselves suggest the field of Multi-Agent Systems,[226,334] but state that this research would *"typically lack the dynamicity required for the knowledge plane"*. But research into cybernetics, self-organisation, autonomous robotics, etc., as well as the adaptation of phenomena and behaviour found in nature, has led to insights which can help realise the necessary capabilities, as we discuss below.

Adoption of the Knowledge Plane in the Literature The purpose of this knowledge plane can be manifold, with Clark *et al.*suggesting fault mitigation, autonomic configuration, overlay network support and intrusion detection. Razzaque *et al.*[300] proposed a cross-layer approach, in which the knowledge plane provides both a node-local view which encompasses information gathered from the individual layers, as well as a global view of information from the network. Kappler *et al.*[180] employ a knowledge plane to coordinate self-organisation in mobile phone networks. Pujolle *et al.*[292] proposed a goal-based networking architecture for wireless networks, which employs situation-aware agents to decide on the protocol configuration. Similar approaches have been applied to sensor networks,[291] IP-based networking equipment,[122] 4G mobile phone networks,[11] and ad-hoc networks.[222,223,224] Pujolle's approach led to a multi-plane system[43] consisting of: a data plane which forwards the packets; a control plane which monitors operational aspects such as throughput, mobility, reliability, or security; a management plane administers the other planes; and knowledge and configuration planes which together form the piloting system. This system uses distributed agents, located on the devices in the network, which autonomously decide when to perform a handover to another access point. For this purpose they incorporate situational knowledge they actively extract from their network neighbourhood and passively receive from other nodes. As part of the Iris framework,[341] Nolan and Sutton present[262] a cognitive wrapper for radio re-configuration, which can *"adopt any feasible game-theoretic, genetic algorithmic, artificial neural networks, Bayesian or Fuzzy system logic"* approach. Knowledge is represented ontological via the OWL language. Their framework design incorporates functionality to re-configure the network stack.[340] While the theoretical direction of this research thus conveys interest in research areas similar to our own, all further publications indicate[91,342] a focus on lower layers, and contains virtually no reference to stack composition, configuration or learning tasks. CogNet[298] provides an overlay network with cognitive features, which

extends[280] the Host Identification Protocol[9] such as to provide improved mobility support.

2.3 Implementing Autonomy – The Autonomic Control Loop

The facilities required by an entity to achieve autonomy are defined nearly identical in all areas of research, i.e. are based on the original definition of the *feedback loop* from Norbert Wiener's seminal *Cybernetics*,[376] which is alternatively called an autonomic control loop[83] or an cognitive cycle,[244] and which is illustrated in *Figure 2.4*. Similar control loops seem to control several aspects of e.g. human behaviour,[68,205,297] but their stability properties are difficult to proof. Dobson *et al.*[83] elaborate on this problem and investigate the possibility of control-theoretically modelling and analysing such systems.

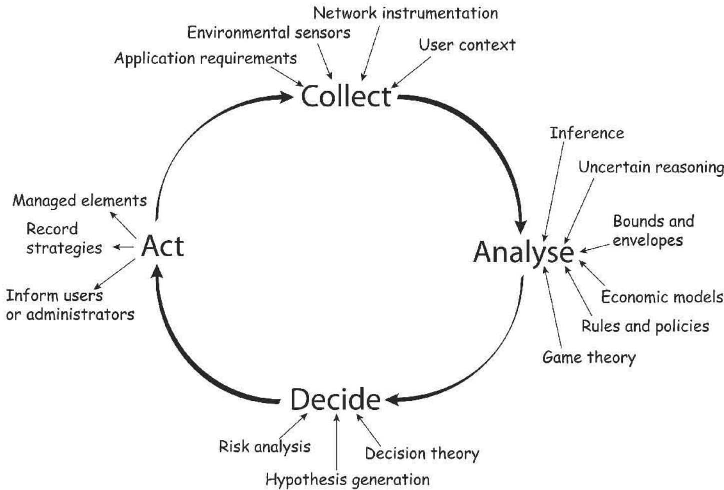


Figure 2.4: *Autonomic Control Loop, from* A Survey of Autonomic Communications.[84]

We group our discussion of related work according to the four aspects embodied in this loop, i.e. the gathering of information in *Section 2.4*, the analysis of this data in *Section 2.5*, the decision process based on this analysis in *Section 2.6*, and the possibilities for executing the resulting decision in *Section 2.7*.

2.4 Collection of Information

The first phase of the autonomic control loop encompasses the collection of information. If one were to compare an autonomic system to a living being, these facilities could be described as its eyes or sensors. As pointed out e.g. by Varghese,[365] the flow of information on the packet-switched Internet is difficult to observe, since the Internet itself was not designed[61] to provide such information in an easily accessible way, and does not maintain state data akin to the per-call data kept on circuit-switched networks. Nevertheless methods like active and passive measurements, information sharing, and experimentation can help us gather a significant amount of information about the network conditions based on external logic.

2.4.1 Measuring Methods & Metrics

Technologies for reporting local measurements are widely supported by common networking hardware, e.g. via SNMP, NetFlow[104] (RFC 3954[59]), or IPFIX (RFC 5101[60]), and measurements can be performed either directly on the routing hardware or on dedicated measurement devices.[71] The resource requirements imposed by measuring are one of its major difficulties. Whereas packet-forwarding functionality can be provided with relatively little state information, such as ARP- or routing tables, detailed traffic measurements require far more memory and processing. One way of reducing the impact of measurements is to summarise the data at the place of measurements, and e.g. only report statistics on a per-flow basis, either via aggregation (packet counting, etc.) or sampling, i.e. extracting only a subset of all packets.

Packet counting is one of the most commonly applied statistical aggregation measures in router hardware and used e.g. for accounting purposes, but managing a large amount of counters for various traffic types is resource intensive. For example, Ramabhadran[293] and Zhao *et al.*[392] propose efficient methods for counter management, which only keep short counters in fast memory and dump them to slower storage at a lower frequency. Another aggregation method has been proposed by Krishnamurthy *et al.*[203] for detecting changes in traffic data based on compact sketches,[255] i.e. projections along pseudo-random vectors of reduced dimensionality.

Sampling can be performed uniformly, e.g. every n-th packet, pseudo-randomly,[274] by assigning them to strata according to an attribute,[58] probabilistically based on their content, e.g. by using a Horvitz–Thompson estimator,[162] or e.g. via one of the sampling functions proposed by Duffield *et al.*.[95] A method based on Max-Min Fair[33] allocation of the sampling budget

has recently been proposed[93] which aims to minimise the average relative estimation error.

Detailed and accurate analysis often requires specialised approaches. For example, to identify the types of flows that cause the highest bandwidth usage over a time of hours or days, Estan *et al.*[105] utilise multi-stage filters. Each filter stage consists of a fixed number of buckets, into which flow data is stored based on the result of a hashing function, which is unique w.r.t. the stage. Only those flows which cause bucket overflows in all stages are stored.

Distributed measurements of high-volume traffic are particularly difficult, because data packets cannot easily be uniquely identified. One method for solving this problem is again the use of hashing functionality. For example, Duffield *et al.*[94] measure the spatial flow of traffic across a network by only reporting a manageable subset of packets, which they select based on the value reported by hashing function. As the same hash function is used on all routers and switches within the domain, a packet that is selected on one link is also selected on all other links as it traverses the network.

Measurements outside the local administrative zone are even more difficult, as access to NetFlow data, etc., is limited for privacy and business reasons. Often the dynamics of intra-protocol traffic are used to derive or facilitate measurements. Even though protocols such as TCP were not designed with measuring in mind, in-band measurements are still possible. TBIT[268] as well as Jaiswal *et al.*'s[175] method, can infer many of the configuration parameters of TCP in a non-disruptive manner. Sting[311] can query unmodified TCP servers and derive packet loss measures along the forward and reverse path. Similarly, the pathload[174] tool estimates the path bandwidth based on the assumption that the one-way delays of periodic packets increase when the send rate exceeds the available bandwidth. ImTCP[210,361] is able to estimate the bandwidth by means of packet rescheduling, and thus avoids network overhead. Bellardo *et al.*[31] provide three methods for measuring packet re-ordering, also by means of TCP. Externally generated connections can also be (ab-)used for measurement purposes. The Sidecar[314] tool, for example, hijacks TCP connections, inserts packets that resemble retransmitted data, and thus avoids problems caused by intrusion detection systems which often misclassify measurement traffic as intrusions. And TCP is only one example of the protocols that can be utilised for measuring purposes.[227] Spring *et al.*summarise[327] some of the methods for measuring and envision an opportunity for *"collaborative reverse-engineering of the Internet"*, i.e. map the Internet's client population, workload, etc.

Measurement accuracy is another problem, as e.g. probing and the computations and traffic related to measurements can introduce a bias into the measurements, which is especially pronounced in multi-user systems,

as e.g. Sommers *et al.*[325] report. They propose countermeasures in the form of a low-level, high-priority measurement subsystem. In systems where such problems can be eliminated, much finer-grained measurements are possible, as e.g. Lee *et al.*[213] method based on LDA[198] allows for end-to-end delay measurement accuracy in the microsecond range even when packet-reordering can occur.

The placement of sensors or monitoring points in the network, as well as the facilities for processing the measurement data, are also very important, especially w.r.t. to scalability and adaptability aspects. Liotta *et al.*[220] present a decentralised solution based on mobile agents (see *Section 2.7.3*), which offers a near-optimal performance w.r.t. to traffic load and high adaptability. As similar approach is followed by Gavalas,[127] while the FLAME[17] platform utilises kernel-level code deployment for monitoring in active networks. Sifalakis *et al.*[316] developed a resource-efficient platform that generates detection automata[160] at run-time based on a formal specification, preserves the temporal order and causality of events, and is specifically designed for autonomous and evolvable network environments. In Ganglia,[232] applications register for specific monitoring functionalities. The affected nodes then monitor their local state and send measurement data via multicast whenever an update occurs. Han *et al.*'s[151] approach uses policies to define monitoring tasks based on the application and network state. These policies encode actions executed at run-time based on the defined preconditions, but the policy specification itself is static. DYSWIS[240] encodes monitoring rules within the nodes, which can trigger queries to their peers in a distributed fashion, for example, whenever its own observations indicate that further investigation is warranted. NetQuest[326] uses network inference based on Bayesian experimental design[218] to select a subset of monitoring rules and parameters such that the information gain is maximised. IBM's Clockwork[307] is a method for constructing autonomous predictive systems based on feed-forward control loops which use statistical modelling, tracking, and forecasting based on an autoregressive time series.

Due to the special problems induced by on-line stack evolution – fluctuations in measurement accuracy and possible communication loss due to experimentation with faulty stacks – we had to develop our own measurement and communication facilities to counter these problems. We introduce these facilities, which integrate and can collaborate with some of the aforementioned approaches for data acquisition, in *Section 5.1* and discuss our communication facilities in *Section 5.3.4*.

2.4.2 Cross-Layer Designs

Sometimes the needed information is already present within the network or the node, but not accessible or readily available. Routers, for example, know when they drop packets do to congestion. If this information is forwarded to the end host via ECN, the TCP instance there can adapt its send rate appropriately, instead of aggravating the problem by retransmitting the supposedly lost packet. Already back in 1990,[65] Clark and Tennenhouse introduced the notion of Application-Level Framing, by which the application decides how the transferred data is to be divided into units for the lower layers to handle. Williamson and Wu's CATNIP[377] encodes context-knowledge such as the size of HTML-documents in the protocol headers, either in the form of a single "priority bit" or through packet sequence numbers, based on which the use and configuration of protocol variants such as Rate-Based Pacing or Early Congestion Avoidance for TCP, or Random Early Detection for IP is controlled. Other cross-layer approaches,[301] which share information between layers, have been proposed e.g. to improve self-organization in wireless networks.[299] Srivastava *et al.*[330] provide an interesting overview of the opportunities and challenges of such designs.

We provide a shared database for cross-layer information sharing and other applications in our design, as discussed in *Section 3.5.3*. We further extend this approach to protocols on the same conceptional[illegible] layer, so that e.g. transport protocols such as TCP or UDP can share the necessary state information to enable replacing them without disruption to ongoing communications as discussed by example in *Section 3.7.2*.

2.5 Analysis of Information

The raw measurement data on its own is of no use until it is processed and the inherent (state) information extracted or converted into a form that can be used for decision making. In a living organism this data processing stage encompasses e.g. those neurons that convert the raw sensory data into an image and associate it with a known object. Based on the situational awareness gained by this stage, the system may decide on how to act, a process we discuss in *Section 2.6*. With respect to analysis, we concentrate on those areas which are important for our research, i.e. how to perform experiment, how to derive a measure of utility or fitness, and how to detect or even predict changes in network conditions.

[illegible] Our system design is actually layer-free.

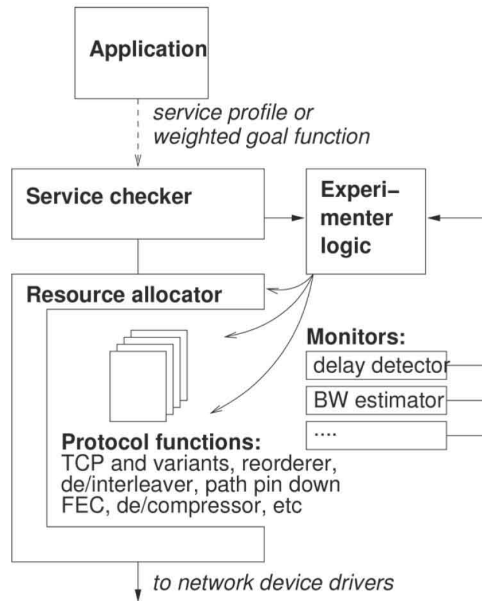


Figure 2.5: *The architecture of the framework for serial experiments.* [296]

2.5.1 On-line Experimentation

One of the fundamental aspects of our design is the concept of trialling possible stack configurations on-line to explore how well they perform under actual conditions. Trial-and-error exploration is one of the most basic and most common methods applied for autonomic problem solving in nature. [382] In artificial intelligence research this method was consequently applied from the very beginning, for example in a model of human thought processes when playing chess. [318] We therefore limit our discussion here to some examples close to our own research.

Zhang [390] analyse the performance of Distributed Breakout [389] and Distributed Stochastic Algorithms [107] (DSA) for the distributed scan scheduling problem in sensor networks which need to limit their communication overhead and lack global knowledge. They come to the conclusion that DSA offers superior performance under controlled conditions. This algorithm is very simple as it basically consists of random selection of a new value if conflicts exist, communication of this value to the remote nodes, and the collection of remote reports, i.e. trial-and-error experimentation.

Within the context of cognitive radio or battlefield communications, the ADROIT[356] project developed a cognitive component[357] which employs a neural network to decide on the most appropriate strategy choice based on the environmental conditions. During a trial run, all of these strategies are tested on-line and used to train the neural net. As this system in total employs only four different and pre-configured strategies and uses 12 input signals, the neural net is able to learn the appropriate choice in around 360 trials.

Ramos-Munoz *et al.*[296] explore how protocols perform by simply deploying them in series or in parallel and measuring their performance. They reason – just as we do – that future networks will become increasingly impossible to model accurately, that gathering sufficient information about the network state becomes ever more difficult, and that simulations will differ from reality, e.g. because *"not all implementations will adhere to the supposed model"*. Their framework, which is shown in *Figure 2.5*, consists of a pool protocol functions, monitor helpers, a service checker, experimentation logic, and a resource allocator. The protocol function pool contains specifications of protocol configurations or complete stack configurations, which are grouped according to their service specification. The monitor helpers gather performance measurements from the network and the system. The service checker decides whether the tested service complies with the application request, based on an user-provided fitness function and the measurements from the monitor helpers. The experimenter logic performs the experiments, i.e. selects when which service or service configuration is deployed in the network stack. The resource allocator executes resource policies such as splitting data streams between two services. The experimenter logic performs experiments whenever the network conditions change or periodically, and selects a known-good protocol and two candidate protocols for testing. It then runs them either in parallel, partially diverting the traffic to one of the protocols, or serially, one protocol after the other.

Bicket's approach[35] for locating the most appropriate bit-rate for wireless communications also applies trial-and-error experimentation, but does not test all possible settings. Instead it limits the experimentation to those bit-rates that promise to offer a higher throughput under lossless conditions and preliminarily aborts trials once multiple consecutive packets are lost. Experimentation is performed infrequently, and the bit-rate that promises to offer the best throughput is utilised, and thus the impact of experimentation on operations is limited.

We discuss the methodology we employ for experimentation in *Section 5.3*, and the means for ensuring comparability between experiments in *Section 5.3.3*.

2.5.2 Goal- & Utility-Measures

For autonomic communication systems to be able to adapt their behaviour towards a defined goal state, they naturally have to be aware of what constitutes a desirable performance, which implies the need for an either explicitly or implicitly defined performance measure. This measure can be e.g. encoded as a machine-level policy description, semantically based on an ontology, expressed mathematically as a fitness function, or expressed in any other way, as long as it enables the system to infer whether the current configuration resembles a goal state or not, or even better its proximity to the goal state.

The problem of how to arrive from a high-level goal description at a precise low-level expression that can be evaluated by the system has been researched e.g. in the context of the semantic web. Strassner *et al.*[337] contest that the concept of the knowledge plane was insufficient to express business goals and to integrate heterogeneous technologies. They propose an inference plan which is based on ontologies and information models to extend the knowledge plane. They explore this concept in the FOCALE project, for which they propose the ontological DEN-ng[338] context and policy model, and which they designed explicitly to support autonomic communications and the corresponding learning facilities. Lewis *et al.*[217] apply ontologies to Content-Based Networks, which are based on the Resource Description Framework (RDF).[208] Zhou *et al.*[393] apply DAML+OIL[161] for QoS provisioning, as do Dobson *et al.*.[82] Similar approaches have also been proposed for Cognitive Radio networks.[28]

Kephart[189] advocates the use of scalar utility measures – derived from the current state of the system or its environment – for self management: "*Utility functions are attractive for autonomic computing because they provide a principled basis for automated agents to make rational decisions.*" Bennani *et al.*[32] and Walsh *et al.*[370] use sigmoid-shaped utility functions based on e.g. response time or throughput to regulate the resource allocation process in data centres or on high-level concepts such as business terms or service-level attributes. Similarly, Kelly[188] reduces multiple decision criteria criteria to a single integer value, and Sutherland uses virtual currencies[97] for allocating resources on a PDP-11. Petrova *et al.*[279] use utility-based goal definitions for learning the utility of spectrum selection and routing issues in cognitive wireless networks. Their utility function definition is based on the expected Value of Perfect Information,[344] to evaluate the potential gain of an action in relation to the cost of performing e.g. an experimental change of configuration. Jon Crowcroft *et al.*proposed a pricing scheme[73] which combines the resource usage of TCP connections with user-assigned weights.

By tweaking the TCP parameters, they achieve fair, selective QoS without over-provisioning, connection acceptance control, reservations or multiple queues on the routers. Of particular interest to us is the research of Lee *et al.*[211,212] who try to translate high-level user preferences into a machine-evaluable utility or fitness measure by means of a model of the user's context. For this purpose they apply a multi-layer neural network to predict a value of the service based on user input for previously encountered situations. They provide an user interface containing a simple "better" button which indicates whether a setting is considered superior to another and thus acts as reward for training the ANN. Our system evolves by means of a fitness function which rewards stack configurations according to their perceived value to the user, where Lee *et al.*'s method could be easily integrated.

Völker *et al.*[368] introduce a meta-architecture, which run on top of instances of underlying network architectures (Netlets), such as SILO, RNA, or ANA (which we discuss in *Section 2.7*), and selects between them based on its presumed utility. For this purpose they utilise multi-attribute utility analysis[186] to aggregate each NetLet's tested utility into a ranking based on the user's preferences.

The relatively recent ChoiceNet[56] also proposes the provision of multiple protocols, stacks, etc.in parallel, between the users choose based on their overall experience: *"Depending on the user's satisfaction* [...], *they continue to use the chosen service or switch to another (i.e., vote with their wallet)"*. The utility thus is directly based on economy, the cost of providing the service vs. the expected and actual monetary gain.

Our own system utilises a dynamically-specified fitness function to determine the utility of stack configurations, which we introduce in *Section 5.3.5*.

2.5.3 Classification & Detection

As the performance of network stacks depend on the situation of the network and the ongoing traffic, we require a reliable method for deciding whether the conditions during different experimentation are sufficiently similar and the results therefore comparable. Due to fluctuations and measurement inaccuracies, as well as the unknown correlation between changes and their effect on stack performance, a simple calculation of a delta of measurement values is insufficient for deciding whether the network state has changed or not. We present several methods for data analysis that we consider promising for the purpose of predicting whether two situations are identical (anomaly detection), and to group data according to their similarity (classification).

Artificial Neural Networks

An Artificial Neural Network (ANNs) is a machine-learning technique which is often applied for filtering, classification, or clustering purposes. ANNs mimic the operation of neurons in the brain, and are trained through application to a set of correctly classified examples (supervised learning). Nodes are often arranged in layers, e.g. one input, one output, and one hidden layer in between, were all nodes on the input layer propagate information to the hidden layer, which in turn propagates the information to the output layer. The number of nodes on the input layer corresponds to the number of input signals, e.g. to the number of measurements, on which the classification process is to be performed, the nodes on the output layer represent the result of the classification process. Propagation of information is realised through connections (vertices) with attached weights, and the value assigned to a node corresponds to the weighted sum of its inputs. These weights are the factors that are actually learned by the network by means of a cost function that defines the distance to the intended solution. Feed-forward Neural Networks forward data only to the next layer, i.e. they represent acyclic networks, whereas Recurrent Neural Networks contain cycles. The performance of ANNs depends strongly on its design, i.e. the wiring, cost function, and number of nodes. If too few hidden nodes are present, the output error is high. If too many hidden nodes are present, the risk of over-fitting, i.e. learning a function that represents the training set very well, but increases the error on unknown input data, increases.[254]

Neural Networks have frequently been applied for anomaly detection tasks. Gonzalez *et al.*[138] present a method for this purpose which is inspired by the immune system and useful when only positive samples are available for training. Estevez *et al.*[221] apply several statistical tests, e.g. the Kolmogorov-Smirnov test, to measure normality of HTTP header features and detect anomalies. ANNs have been successfully applied for anomaly detection, for example by Kozma *et al.*[200] who use a three-layer neural network to detect weak anomalies in sensor measurements of nuclear power plants. Several approaches for adaptive state filtering are presented by Parlos *et al.*,[270] who use Recurrent Neural Nets to approximate the non-linear dynamics of the filter. The resulting method is thus applicable to real-world problems, but still requires an initial model, even though this model may be somewhat inaccurate. Haigh *et al.*[148] use an ANN to learn a model of the fitness landscape in a MANET configuration setting. Due to the huge size of the search space, they utilise existing analytical models and only learn the error within these.

Neural networks are also used to learn e.g. the optimal configuration of

radio parameters, as we described in *Section 2.5.1*. Gelenbe and his group utilise so-called smart packets for many different networking purposes,[130, 131] for example for adaptive routing and QoS provision.[132,133] These packets utilise Random Neural Networks[129] deployed on the routers throughout the network. Each network is composed of as many neurons as there are outgoing links, and the corresponding weights are reinforced based on their contribution to the success or failure of achieving the QoS target.

Clustering

Clustering algorithms perform unsupervised classification by grouping data samples such that the overall dissimilarity between group members are minimised. Clustering has many important applications in data mining, for example for traffic analysis. k-means is an example of a kernel-based algorithm that is in common use and which we introduce in *Section 4.2.2*, and which e.g. Lakhina *et al.*[206] apply for classifying anomalies in traffic data. k-means requires the number of clusters to be specified in advance, but methods such as X-means[277] can work around this limitation. Hierarchical clustering is another class of clustering methods, and operates by building a hierarchical structure of the data according to its proximity matrix. These methods need no a-priori knowledge of the number of clusters, which often is advantageous. Dornbush *et al.*,[88] for example utilise Hierarchical Agglomerative Clustering for clustering vehicles according to their position in VANETs. The performance of clustering algorithm depends very much on the type of problem it is applied to, as Xu *et al.*[386] show in their comprehensive survey of the field.

2.6 Decision

The decision on how to act based on situational awareness is an operation which is usually associated with cognitive brain functionality and as such arguably the most difficult task of our system: It has to devise candidate stacks, which promise to offer better utility than the known solutions, based only on the measured utility of these. We focus in our discussion on on-line learning techniques, which do not require a world model and employ exploration of the configuration space. The reason for this restriction lies in the nature of the problem we intend to solve. The sheer size of the search space and the relative lack of design-time knowledge about this space and utility distribution therein biased our focus towards probabilistic and bio-inspired methods. Expert systems and other model-based reasoning methods[308] infer a

conclusion based on a world model. To build such a model, sufficient knowledge of the application environment and problem space – e.g. the deployed protocols – is needed, which does not fit our requirements for autonomy. Supervised learning[248] requires labelling by an expert, which is contrary to our aim of reducing the administrative overhead. We further expect our system to be deployed in unknown situations for which no appropriate training data is available. Unsupervised methods such as clustering are very useful for data classification as they can find hidden structures embedded in the data, and are therefore discussed in *Section 2.5.3*. But since these methods do not explore the search space for new candidate solutions to trial, and also do not utilise a reward or utility measure to evaluate these candidates, we cannot easily integrate them into our stack evolution approach.

2.6.1 Evolutionary Algorithms

Evolutionary Algorithms are meta-heuristic optimisation techniques operating in accordance to the principle of biological evolution: *"given a population of individuals, the environmental pressure causes natural selection (survival of the fittest), which causes a rise in the fitness of the population over time"*.[99] Genetic Algorithms operate on populations of (often binary) strings, the chromosomes, each of which encodes a configuration in the search space. Genetic Programming encodes solutions as computer programs, whereas Evolutionary Programming encodes only the parametrisation of these programs, and keeps the program itself constant. We describe our own implementation of an Evolutionary Algorithm in *Section 4.1.3*.

Eiben[99] proposes the use of Evolutionary Algorithms for autonomic computing and presents an exemplary application for autonomous load balancing of a web service provided by several servers in the network. Montana *et al.*[249] apply Genetic Algorithms for re-configuring network topologies, e.g. wireless point-to-point links between mobile nodes, depending on traffic patterns, resource availability and other dynamically changing requirements. They encode a list of variable length which consists of 3-tuples representing the source node, destination node and number of channels that connect these two nodes. Three genetic operators are used to create children: crossover, where the 3-tuples of both parents are randomly combined into one list excluding those tuples which violate constraints, local mutation, where the number of channels is increased or decreased by one, and global mutation, where between half and all-but-one of the tuples from one parent are attributed to the child and the remainder filled up with random tuples. Populations are kept duplicate-free, and roulette-wheel

selection is used for parent selection. Yamamoto *et al.*[387] utilise Genetic Programming to autonomously synthesise protocol functionality from fundamental building blocks. Their research shows some of the difficulties of truly autonomous code generation, as their programs often cheat, for example by reporting incorrect transmission statistics, to achieve a higher fitness than appropriate.

Evolutionary Algorithms can also be realised as agents distributed across a network. For example, Laredo *et al.*[207] evaluate distributed Evolvable Algorithms placed in a simulated peer-to-peer network by measuring their performance, i.e. solution quality and algorithm speed, through application to artificial fitness landscapes. Similarly, Genetic Algorithms have been used for QoS-based routing e.g. by Barolli *et al.*[29] or Xiang *et al.*,[385] and for shortest path routing by Ahn *et al.*[6]

2.6.2 Reinforcement Learning

Reinforcement Learning is a class of trial-and-error learning methods which focusses on maximising the overall sum of a scalar reward signal:[343] Based on sensory input from the environment, an agent chooses an action to perform. The rewards communicated by the reinforcement signal are not immediate, but depend on the value of the state the agent is currently in, thus actions can trigger delayed benefits some time in the future. Reinforcement Learning significantly differs from the supervised learning used to train e.g. ANNs, as the agent has to find an effective trade-off between exploration of unknown states and exploitation of states which are known to offer a positive reward. Many methods in this category model a finite Markov Decision Process[343] (MDP) in which the probability of reaching a specific state depends only on the previous state and the performed action. Thus the expected reward is the sum of the rewards for all the states multiplied with the probability of reaching that state. One of the most common methods is Q-learning,[373] but many other effective algorithms exist, some of which are surveyed by Kaelbling *et al.*[178] and by Busoniu *et al.*[44] for application in multi agent systems.

Reinforcement Learning techniques have been applied, for example to routing by Dowling *et al.*, whose SAMPLE[90] algorithm for MANETs collaboratively learns the most effective route based on positive and negative feedback. Their MDP-based learning method overcomes the need for complete observability by favouring recent observations using a finite history window and reducing the effect of older values by means of a decay model similar to the one used in Ant Colony Optimization.[86,87] Similarly, Sridhar *et al.*[328] apply Reinforcement Learning for power-usage-optimisation in sensor networks, based on a reward function that weighs power drain against correct

sensing of temperature changes, and thus achieve an extension of the sensor's operational life.

2.6.3 Swarm Intelligence

Swarm Intelligence refers to the collective behaviour of decentralised multi-agent systems. These agents co-ordinate their behaviour through local interaction with each other and their environment, using algorithms which often were inspired by or modelled after swarm-building animals.

Swarm Intelligence is often employed for distributed optimisation tasks in networking. In particular, the behaviour of ants, i.e. their probabilistic exploration, the marking of effective routes towards food sources with pheromones, and the selection of routes biased towards thus marked paths, was utilised for routing,[80] peer-to-peer networks,[23] distributed computing,[353] and the semantic web.[252] All mentioned approaches utilise variants of the Ant Colony Optimization[86,87] algorithm. Sesum-Cavice *et al.*[313] exploit a somewhat different method to organise a Peer-to-Peer network for efficient information retrieval. The behaviour of other swarm-building animals such as fireflies was also effectively appropriated for networking processes, e.g. for clock synchronisation in ad-hoc[363] and sensor[369] networks.

While many of these algorithms operate in a distributed fashion, the study of the behaviour of honey bees has let to the development of powerful optimisation algorithm,[182] which compares favourably with techniques like Genetic Algorithms or Evolution Strategies.[181] The algorithm employs three types of artificial bees – employed, onlooker, and scout bees. There is one employed bee for every known food source, i.e. known location in the search space. The employed bees communicate the location of the food to onlooker bees, which choose one of these, and go to explore another randomly-selected location in the vicinity of the chosen food source. Scouts randomly search for new food sources. Old known food sources are over time replaced with new ones found by the scouts depending on the amount of food, i.e. fitness value, present there.

Cuckoo Search[371,388] is a relatively new and rather effective[123] optimisation technique modelled after the brood parasitic behaviour of cuckoos. Every cuckoo lays one egg into a nest, i.e. location in the search space, which it chooses randomly using Levy flights.[230] If that new location's fitness is higher than the old one, it is replaced, otherwise the old location is kept. Additionally a fraction of the worst performing nests is replaced by randomly chosen new locations.

2.6.4 Other Approaches Inspired by Nature

Whereas interest in autonomously interacting communication systems became widespread in the last decade, the principal idea of mimicking natural phenomena which exhibit self-⋆ properties itself dates back to the early days of cybernetics, as e.g. Ashby[19] himself already explored the possibility of self-organising systems. We limit our discussion here to some more recent examples of nature-inspired approaches, but many other approaches for networking exist, as surveyed by Dressler *et al.*,[92] and Babaoglu *et al.*[22] even provide a set of design patterns for distributed computing based on biological approaches.

Modelled after interaction features on the cell-level, Leibnitz *et al.*[214] utilise the concept of attractors[184] for multi-path routing. Here the probability of selecting a specific path for an outgoing packet is relative to the concentration of nutrient molecules on the path, i.e. the packet delivery ratio along a path. The combination of attraction and inhibition in turn was used e.g. for the configuration of sensor networks,[259] where nodes stop sensing when adequate sensor coverage in the vicinity is guaranteed, i.e. the inhibitor concentration is high enough. The interaction of cells that leads to the bristle differentiation of Drosophila flies was modelled by Tateson *et al.*[346] and used to generate a channel allocation plan for mobile phone networks. Ganguly *et al.*[125] present a search algorithm for peer-to-peer networks which utilises a concept of proliferation based on reaction-diffusion in a similar way to the humoral immune system. Artificial Immune Systems[163] themselves have also been used for networking purposes, such as the detection of misbehaviour in ad-hoc networks.[310]

The interaction of forces and particles has also been successfully applied for the dissemination of information. Artificial chemistry has been used for networking protocol design,[239,360] and Stoy *et al.*[335,336] propose an amorphous self-reconfiguration algorithm, based on Abelson *et al.*'s[4] research, which consists of a coordinate propagation mechanism, a gradient generation mechanism, and a mechanism to relocate modules, through which a randomly configured robot can recompose itself to resemble a target shape. The gradient generation mechanism operates by locally disseminating an artificial chemical to attract other nodes to its neighbourhood. Likewise computational force fields have been used to coordinate the motion of mobile agents.[229] In this model, agents can generate artificial force fields which are propagated through the embedded infrastructure. These agents thus convey contextual information across the environment, which can be used by other agents by simple following the field gradient, e.g. for locating a target or avoiding crowds. Their TOTA-approach[228] represents contextual in-

formation by means of distributed tuple structures in the network, which it reshapes dynamically in accordance with the dynamics of the network.

2.6.5 How to Choose and Configure an Optimization Algorithm

In their seminal 1997 paper,[381] Wolpert and Macready analysed how so-called black box optimisation algorithms, such as Evolutionary Algorithms or Simulated Annealing, perform for different problem classes by looking at a-priori constraints as well as through calculation for specific problem sets, and came to the conclusion that any advantage one (class of) algorithm might have for a specific set of problems was counteracted by a inferior performance for other problem sets, i.e. worse performance than random search on average. In particular, they show that *"for both static and time-dependent optimisation problems, the average performance of any pair of algorithms across all possible problems is identical"*, and that this performance depends on how well the probability distribution of the possible solutions the algorithm visits fits the underlying distribution of the problem. Consequently, if the probability distribution for a particular problem class is unknown, one cannot easily predict which algorithm might perform best.

The performance, i.e. the quality of the found solution and speed of the algorithm, of a specific class of algorithms for a specific problem furthermore depends on the parameters used to configure it. For example, Eiben, Smit, *et al.*[30,102,201,320,321,322,378] analyse this problem in detail, and present techniques for on-line and off-line parameter optimisation. Eiben *et al.*[101] also show that the on-line adaptation of normally ignored parameters, such as population size or selection operators, can further improve the performance. Particularly interesting in this respect are meta-optimisation techniques, such as Eiben *et al.*'s[100] approach of applying Machine Learning methods to learn the optimal parametrisation of Evolutionary Algorithms. Mori *et al.*[250] also utilise a two-layered meta-heuristic for adaptation. They apply a Genetic Algorithm to evolve individual populations – islands[375] – between they select by means of an environment-identifying ANN. Thus the individuals on each island evolve independently and specialise towards the environment on their island.

2.7 Action

Once a decision has been taken, the plan has to be put into action. Just as a living being might move its legs to reach a destination, the autonomous system has to possess actuators through which it can influence its surroundings

or itself. In this section we therefore introduce technologies, designs, and architectures that can enable the system to act, i.e. to re-configure itself or influence other entities in the network to do so.

2.7.1 Protocols & Stacks

Throughout *Chapter 3* we discuss our own design for flexible micro-protocol-based stack composition and configuration, which is heavily influenced by and based on the foundations laid by the research we present below. In *Section 3.4* we introduce how the network stack in our system is composed and configured at run-time out of micro-protocol modules and how the composition process is defined and constrained by means of a specification ontology and in *Section 3.7* we introduce the interaction between these modular stack components and the control flow between them.

Protocol Configuration & Tuning

Stack functionality can not only be controlled through the stack's composition, but also via the configuration of the protocols encompassed therein. Current implementations of TCP, for example, export many configuration knobs that are accessible through e.g. the sysctl interface in FreeBSD. The administrator can thus select whether to enable Appropriate Byte Counting (RFC 3465[12]) congestion control, etc. Several protocols have been explicitly designed with re-configurability in mind. The Fully Programmable Transport Protocol,[106] for example, selects the employed rate, congestion, and error control methods based on the configured QoS requirements. Robles *et al.*[305] describe a location protocol for wireless sensor networks, which an application can adapt at run-time to match its requirements. Gu *et al.*[142] allow applications to provide their own congestion control algorithms in user-space. Jaganathan *et al.*[173] provide a customizable transport protocol for application in FPGAs.

The combined tuning of multiple protocols across layers can be particularly rewarding, and has for example been used for parameter control in MANETs[136,148] or Cognitive Radio applications as introduced in *Section 2.1*. Several examples of architectures for this purpose are also presented in *Section 2.2.2*.

Protocol re-configuration does not necessarily require support by the existing protocol implementations. Feldmeier *et al.*[109] suggest the use of protocol boosters, i.e. protocol modification modules, which transparently adapt the corresponding protocol's operation such that better performance is achieved. For example, TCP performance on asymmetric channels, where

the return channel is far slower than the forward channel, can be boosted by adding a module on the receiver side which ensures that only the most recent acknowledgement is kept in the transmission queue.

A complete network stack can even be constructed by repeated application of a single, flexibly configurable protocol. Touch *et al.* [355] developed the Recursive Network Architecture RNA, which applies a single, tunable protocol template on different layers of the protocol stack. They note that many new protocols include redundant functionality, such as state establishment or virtualization, which are already provided by other layers. Their results are threefold: Firstly, services are relative to the layer they are provided on, e.g. link layer security is only sufficient for the link layer, network layer security only for the network layer. In other words, end-to-end services cannot be provided by hop-to-hop services. Secondly, many of the redundant services offered by the multiple layers could be avoided. For example the redundant error detection currently provided on many layers, e.g. checksumming by IP, TCP, and on the link layer. Thirdly, cross-layer coordination eases operation, as many features are dependent on characteristics or provisions of other layers.

From Monolithic Protocols to Stack Building Blocks

The aforementioned problem of redundant functionality can be solved by decomposing the protocols into re-usable modular components. The modularisation of protocol functionality is particularly interesting since most common protocols such as TCP can be decomposed into minimal functional blocks. [34,152,196] The resulting atomic modules are re-usable by other protocols and can be recombined as needed. Stack designs based on such micro-protocols have been proposed for more than twenty years, e.g. in O'Malley *et al.*'s [267] Dynamic Network Architecture.

Such decomposition is possible because protocol functionality generally follows the same communication paradigms. Karsten, [183] for example, studied the de-construction of communication protocols into fundamental axioms. These axioms can be used to model and formally analyse protocol behaviour and – via meta-compilation – to generate C++ protocol implementations. Chiang [53] developed a mathematically-thorough analysis of the network stack as a whole, and describes how protocols can be systematically designed as distributed solutions to a global optimisation problems.

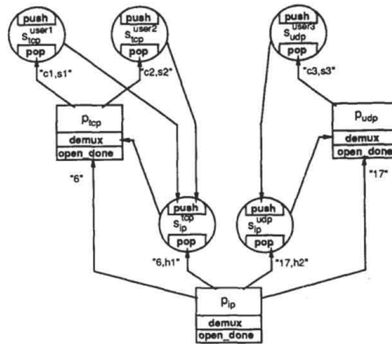


Figure 2.6: *A composable network stack of the x-kernel.*[166]

Dynamic Composition

Modular architectures which provide the necessary facilities for dynamic composition at run-time have a relatively long history: Dennis Ritchie[303] introduced the flexible, coroutine-based stream I/O subsystem for character devices into Unix already back in 1984, which allows the output of one device to be connected to the input of another via pipes. One of the key concepts of this system are *stackable layers*, which Guy *et al.*[145] copied for their Ficus file system and explain as: *"A stackable layer is a module with symmetric interfaces: the syntactic interface used to export services provided by a particular module is the same interface used by that module to access services provided by other modules in the stack. A stack of modules with the same interface can be constructed dynamically according to the particular set of services desired for a specific calling sequence."*

Two of the most important early contributions to the research on dynamic network stack composition were made back in 1991. Hutchinson *et al.*[164] proposed the x-kernel, a operating system kernel which allows runtime construction and composition of networking protocols, and provides abstractions for common protocol functionality. This scheme utilises a hierarchical naming scheme similar to Unix paths to specify the route to take through the stack when decoding an incoming packet, as shown in *Figure 2.6*. Protocols only need to support very few basic operations, open to attach to a lower-layer protocol, push to forward an outgoing packet to a lower layer, pop to retrieve incoming packets, and demux to de-multiplex incoming packets between multiple attached higher-layer protocols. As the x-kernel makes no assumption of the order

or composition of the stack, run-time addition of protocols is possible. These protocols do however need to specify the protocol they wish to bind to at compile time. Tschudin's[358] Flexible Protocol Stacks removed this requirement, thus enabling the free re-structuring of the protocol stack at run-time by means of COMSCRIPT,[253] a composition language modelled after PostScript. Plagemann *et al.*'s Da CaPo,[281,282] composes and configures protocols through a specification language. Apart from the properties of the protocols, application-specific communication requirements and their weights are encoded in this specification, and the stack configuration that best fulfils the combined weighted requirements is determined based on predetermined influence estimates that are part of the property description. Zitterbart *et al.*[394] propose a functional-composition system which includes validity scope checking and composition based on automatically generated dependency graphs.

Beyond (Layered) Stacks

The run-time-configurable Horus[302] group communication system broke the hierarchical stack model in 1996 by stacking micro-protocol modules into different communication groups. These modules each have separate responsibilities, i.e. provide a distinct subset of the protocol functionality, and communicate via an uniform interface. Since the layering paradigm which rules[61] the Internet's design is often, and in most cases for good reasons, compromised, Braden *et al.*[40] proposed to remove layers altogether and replace them with a non-hierarchical role-based architecture. Roles are similar to micro-protocol functionality and encompass e.g. such tasks as fragment, encrypt, or compress. Another relatively recent approach is Dutta *et al.*'s SILO,[96] which also represents a layer-free architecture. As the name suggests, their architecture is based on silos of services, which are assembled on demand according to the requirements of the application and environment. Their architecture also uses micro-protocols, but the interaction between these is regulated by well-defined precedence constraints, i.e. compression can only be placed before encryption, not after. Silos are composed by means of a fairly simple ontology-based non-polynomial algorithm.[366] Service-oriented architectures are explored by Wolf,[379] SOA4All,[85] etc.

Composition Frameworks & Network Architectures

Several of these designs have evolved over time into complete frameworks for protocol composition, some of which Gazis *et al.*'s[128] survey describes

in detail. Touch expanded the concept of recursive blocks he proposed for RNA towards a full network architecture called DRUID.[354] In DRUID, recursive blocks encapsulate the protocol functionality, translation tables provide the name resolution functionality, i.e. they resolve the destination to which information is to be forwarded similarly to the protocol graphs of the x-kernel or Click,[197] and a persistent state state storage. The Configurable Transport Protocol[41] is constructed using Cactus, and has been applied e.g. for grid-computing purposes.[383] El Baz *et al.*[103] also use Cactus to dynamically generate an application-specific protocol for peer-to-peer applications at run-time. Horus in turn led to Ensemble[156] and later to the Appia[243] framework. Mena *et al.*[238] provide an exhaustive comparison of Appia and Cactus and propose several design improvements.

The Pandora[272] project stacks software components which communicate through message exchanges, and can re-configure itself according to the configuration specified via a reflexive interface. One application is the C/SPAN[266] project, in which a flexible web cache called C/NN tunes Pandora according to the measured disk space, request rate, etc., while Pandora in turn configures C/NN according to the observed traffic patterns. Condie *et al.*'s P2[69] also assembles specialised transport protocols using reusable data-flow building blocks. Perhaps most importantly, they present convincing use cases for composable protocols within the context of the current Internet: They argue that the requirements of peer-to-peer applications for routing and buffer management on the application-level, aggregation of congestion state, etc., are far more heterogeneous than those of other applications and require adaptation to the network as a whole, which renders monolithic protocol designs insufficient.

Technologies such as CORBA[251] also offer features very similar to those provided by the protocol stack composition frameworks described above. For example Crane *et al.*[70] and Vandermeulen[364] discuss composable and configurable protocols which operate within a CORBA environment on top of TCP or UDP. The Stratos[134] project considers composition for virtual middleboxes in a cloud environment.

Many Future Internet research projects include architectural provisions for composition of services, protocols, stacks, etc. The ANA[16] project, for example, compartmentalises network entities that utilise the same services or functional blocks and connects them via the so-called IDPs, which we introduce in *Section 2.7.2*. IDPs can be transparently rebound at the discretion of a compositional framework and functional blocks communicate with each other exclusively via IDPs, thus run-time rebinding of the communication paths between functional blocks is feasible.[38] For this purpose ANA includes a functional composition framework[317] which abstracts the lookup

and routing process. The 4WARD project[3] provides a meta-architecture, in which ANA, SILO, etc., can operate in at the same time.[367] ChoiceNet[56] envisions an architecture where multiple alternative stacks, protocols, etc., are offered in parallel, and between which the users decide based on economical considerations.[380]

Conditional Configuration & Branching

Depending on the application, user demands, or network conditions, different stack configurations might offer the best possible performance, thus the selection of a different sub-stack based on a per-flow or per-packet decision may be advisable. The BSD packet filter[234] is remarkable because it allows packets to be captured based on a run-time-defined, and just-in-time compiled, definition of criteria, e.g. value ranges in the packet header. Similar packet-based functionality has been integrated into software firewalls, e.g. ALTQ[55] for queuing and dummynet[304] for network simulation.

The Router Plugins[74] architecture provides per-flow stack composition, as it allows plug-ins to be dynamically loaded into the router operating system's kernel, configured, and assigned to specific flows. The Click[197] router also offers per-packet or per-flow branching, and run-time re-configuration, but modules can only be added at compile-time. The OpenFlow[235] project enables the dynamic specification of rules for controlling the forwarding of packets and flows within the router hardware. Thus two possibilities for flow-based or packet-based sub-stack composition are made possible: Firstly, the handling of forwarding actions can be delegated to a remote router, separate from the node responsible for the composition. Secondly, a router can be instructed to forward packets to different nodes or ports, on which independent (sub-)stacks are provided.

2.7.2 Dynamic Resolution of Names & Functionality

The stack components in our system depend on a underlay-independent means to resolve the location of services and stack functionality. In our system, modules and users should not know whether a service or network entity is reachable via a particular TCP port at a specific IP address, and instead use an generic way of identification. Static associations with particular protocols might break whenever the stack is reconfigured, as the service in question could all of the sudden be reachable via UDP directly over Ethernet instead.

The legacy approaches for naming and addressing on the Internet are inadequate for such dynamically-layered or layer-free architectures in which new protocol services can be added at run-time. In the Internet, service iden-

tifiers are mostly statically assigned, e.g. IANA defines that service type 6 in the IPv4 header identifies TCP. Even theoretically flexible services like DNS or ARP are protocol-dependent as they expect to understand the address formats that are used for mapping. Several more-or-less incompatible visions and proposals for more flexible future naming schemes have been proposed in the last decade.

Balakrishnan *et al.*[27] argue that the current Internet's scheme of only providing one layer of name resolution, namely DNS, is insufficient, that names should bind protocols only to aspects of the underlying structure that are actually relevant for their operation, that persistent names should not impose arbitrary restrictions on the named object, that names should be delegable to other entities, and that the resolution process should include the possibility of specifying sequences, e.g. for source routing. They propose four layers between which resolution is performed, from user-level identifiers over service and entity identifiers to IP address resolution.

Stoica *et al.*[332] propose the Internet Indirection Infrastructure *i3* to generalise the Internet's abstraction of point-to-point communication. In *i3* packets are no longer addressed to a specific end host, but instead to an identifier stored in the overlay networks location service, e.g. a DHT service like Chord.[333] Each identifier is mapped to one node in this network, which is responsible for forwarding the data to the corresponding IP address(es). *i3* thus supports unicast, multicast, and anycast packet delivery, as well as node mobility.

Contrary to *i3*, Crowcroft *et al.*'s[72] *Plutarch* aims to promote network heterogeneity through a *catenet*[3] similar to the one that constituted the original Internet. For this purpose they introduce the concept of a *context*, in which a mapping between names and addresses is defined. For example, nodes in a sensor network can thus map an address within their own context, which refers to a gateway that forwards the data to a host on the Internet. *Plutarch* defines a strawmen API for registration and lookup purposes. A similar architecture has been deployed[169] on the PlanetLab[57] infrastructure to connect networks based on otherwise incompatible technologies such as DTN[108] with the Internet.

Alternative addressing schemes have also been used to improve packet processing. Chandranmenon *et al.*[50] propose the use of a source hash, i.e. a flow identifier, to speed up the handling of packets. They argue that processing power is more expensive than networking bandwidth, and that fields for connection identifiers, network or data link addresses, etc., should be added to protocol headers for this purpose. Tschudin *et al.*[359] introduced Network

[3] Concatenation of disparate networks

Pointers, which encode packet processing functions that are used to resolve local selector labels instead of IP addresses in I^3. This design philosophy was further explored by the ANA project,[16] which utilises the so-called information dispatch points (IDPs) to connect functional blocks, e.g. services or protocol implementations, with each other. Access between functional blocks is abstracted by means of IDPs, which may be transparently rewired at run-time for re-composition purposes (see *Section 2.7.1*). Compatible functional blocks, however, are grouped in compartments and may alternatively use arbitrary addressing and naming schemes between themselves. Name resolution uses a publish / resolve model similar to *Plutarch*. Names consist of context and service fields, and the compartment in which the name is to be resolved is selected by means of regular expression search applied to the context field.[38]

We discuss the specific abstractions for naming and address resolution we use in our system, as well as the reasoning that guided this design, in *Section 3.7.3*. The system itself is however flexible enough to host other schemes, e.g. those introduced above, without modification, as discussed in *Section 3.7.2*.

2.7.3 Run-time Code Deployment

Local protocol deployment at run-time is a well-established functionality, as most current operating systems support dynamically loadable kernel-modules.[304] Thekkath *et al.*[349] and Maeda *et al.*[225] introduce methods for deploying new protocols in user-space, as does the TUN/TAP[202] driver which is available for most common operating systems. Other approaches such as Plexus[113] even allows applications to specify their own protocols written in a type-safe specification language which are installed directly in the operating system kernel.

But the manual effort needed to deploy new and improved protocols and services when they become available is still rather large. As autonomous networks are supposed to reduce the necessity of operator intervention, several research projects explored how to alleviate this and other problems by autonomously deploying code in the network.[159,312,374] One interesting application of active network functionality is STP,[273] which enhances TCP by means of mobile code. STP distributes protocol extensions, written in a type-safe version of C, out-of-band to the network nodes. These untrusted extensions do not require user applications to be modified, are run in a sandbox and at low enough priority to supposedly guarantee reliable communication and TCP-friendliness even in the presence of adversaries. In wireless sensor networks, active code has been deployed by Levis *et al.*[215] and others.

Our system does not directly implement such technologies, but can load the thus deployed protocol code into the system at run-time, as we discuss in *Section 3.7.*

2.8 Summary & Conclusion

In this chapter we introduced the research and technologies most closely related to our own work, and which served as the basis for our own exploration of autonomous stack evolution. We claim that some important aspects of the field have not yet received the attention they deserve, and that several parts of what is needed for a truly autonomous stack composition and configuration system are still missing. For example, past research on stack composition aimed at interoperability in multi-protocol environments, path adaptation to user demand for flow services, cross-layering, and easy adoption of new protocols. But research into the heuristics or logic that would enable truly autonomous adaptation to the – often arbitrary and thus unpredictable – demands of the users, and how they can be applied in a realistic setting, is still very limited. We hope that our own work, which we introduce in the following chapters, can serve as a further step towards filling some of these gaps.

Chapter 3

Rationale and Architecture

In this chapter we introduce the concepts and the reasoning behind our research into autonomous stack evolution, and describe the demands and requirements which we plan to fulfil. Afterwards we introduce the system architecture which we derived from this reasoning, detail the requirements which incorporate our demands for the system, and explain how our architecture reflects those requirements.

3.1 Towards Autonomous Stack Evolution

In *Chapter 1* we introduced the motivation that stimulated our research, based on which we now detail the reasoning that guided the design of our stack composition system. We envision a self-adapting system that encompasses the network stack and realises autonomous stack evolution as follows. **Instead of pre-defining one generic stack for all possible application environments and network conditions, use a stack that is optimised for the current situation. Instead of fixing the stack configuration at design- or deployment-time, adapt it autonomously towards the current needs. Instead of incurring prohibitive costs for analysis of the network and traffic conditions, followed by the design and implementation of a customised stack, let the system itself decide, not based on assumptions, but through actual trial-and-error experimentation. Instead of disruptive replacement of the complete network infrastructure, let it gradually, but persistently and continuously adapt towards a close-to-optimal configuration. Instead of defining the optimum, let the users of the system or the deployed applications define what is considered good.**

The properties and system requirements we deem necessary for such a system are presented in this section.

3.1.1 Continuous Optimization

The first condition that we require our system to fulfil is a continuous, ongoing optimisation process. If the state of the network changes, our system has to recognise this change and react appropriately. If the current performance of the system is inappropriate, it should try to improve its modus operandi. Since we assume that the environment and the requirements of the users are non-static and unpredictable, constant re-evaluation of the performance and adaptation as needed appear to be essential and inevitable.

3.1.2 Situational Awareness

Because we assume unpredictable changes in demands and network conditions, the system has to be capable of independently evaluating the conditions of its operation environment. This implies the autonomous gathering of information about the state of the system, ongoing traffic, network conditions, and user demands, as well as an analysis of the obtained data. The system has to be able to reliably and autonomously perform these tasks, as otherwise the induced operational overhead would nullify the system's benefits.

3.1.3 Flexible Goals and Environments

We intend the users of the system to be able to define the criteria for optimisation at run-time, even if they do not understand the intrinsics of how these goals can be achieved. Applications shall therefore be able to indicate how satisfied they are with the capabilities of the stack, as they are in a better position to retrieve this information from the users. The system has to be able to assess the performance of the stack based on these reports. We thus demand support for an arbitrarily wide variety of independent and underspecified optimisation goals, application scenarios and network configurations, the characteristics of which can be unknown at design and deployment time. Since the composition and configuration of the ideal stack under such conditions can be assumed to be equally unknown, we require the system to be able to create and experiment with new stacks and thus learn how to optimise the stack at runtime.

3.1.4 Gradual and Local Adaptation

As we discussed before, drastic changes, e.g. clean-slate approaches, are far harder to implement in practice than gradual changes. We therefore venture that an autonomous stack composition system should be able to introduce changes on a local basis, i.e. that the network stack of only a few nodes in the network should be optimised at a time. This naturally implies that backward compatibility has to be guaranteed, and that nodes which utilise different stack configurations nevertheless have to be able to communicate with each other. While only one stack is to be deployed on a node at the same time, it may contain conditional branches which lead to different sub-stacks. Since different nodes can be utilised in different ways and be exposed to and participate in different traffic scenarios, their stacks should further be optimised individually, which implies that interoperability between different configurations should not only be possible, but assumed as common and implemented such that it does not impede the stack operations. Our demands further imply that the stack can be expected to operate reliably. For this purpose we require the use of a known-good or baseline stack configuration for most of the time and that the negative effects caused by experimentation with new stack configurations is limited.

3.1.5 Situational Memory

While the environment in which the system is used cannot be assumed to stay stable, sufficiently similar conditions might recur multiple times. An additional requirement therefore is the ability to classify the current situation, i.e. the current state of the network, the traffic conditions, etc., to memorise which stack configuration exhibited the best performance under each of these classes of situations, and to use the appropriate class for operations and stack evolution. Through this requirement we increase the convergence speed and guarantee that the stack composition system can adapt to multiple environments, i.e. different optima, at the same time: As a separate set of stack configurations and utility measurements is stored per situation, the best configuration can be immediately selected whenever similar conditions are re-encountered.

3.1.6 Distributed Multi-node Optimization

As already mentioned, nodes shall be able to individually optimise their configuration. At the same time, however, it shall optionally be possible to maximise the overall network performance, if so desired, without requiring a

centralised decision making entity. It shall therefore be possible to express global optimisation goals locally, with local optimisation leading towards a shared optimal state. Independently controlled optimisation efforts in turn require measures to prevent oscillations and strong fluctuations of performance in the network as a whole, such that e.g. experimentation on one node does not harm similar efforts or normal operations on another node.

3.2 Concepts & Features

Based on the goals defined above, we now elaborate on the concepts that guided our design.

We intend for our system to be able to optimise its operations based on the demands and expectations of the users and applications and depending on the current network and traffic conditions, neither of which are known in advance. For this purpose we employ on-line trial-and-error experimentation: Our stack composition system continuously explores new network stack configurations,[1] trials them on-line by exposing them to the ongoing network traffic, and then evaluates their utility based on user- and application-defined criteria for rating the system's performance.

We modelled the adaptation process after nature and try to evolve new stack configurations based on the performances of previously tested stacks: New stacks for the experiments are devised algorithmically by the evolution logic based on the results of previous experiments. Similarly to the survival of the fittest in nature, this logic favours stacks that exhibit better performance and is more likely to use them as basis for new stack configurations.

Our performance measure is the utility or fitness of the stack. The system collects data pertaining to the trialled stack's performance from the running applications – and hence indirectly from the users of the system – which is condensed into a scalar fitness value. This aggregation is performed according to an administrator-defined fitness function. Conceptionally, this fitness value expresses how closely the current system performance resembles the expectations of the running applications and of the users of the system, and thus determines how likely a stack is to "survive".

The system autonomously assesses whether exploitation of the current stack or exploration of new stack configurations is most beneficial: Just as hungry predators will roam further for their prey, the intensity of experimentation depends on the overall achieved fitness: A high fitness value –

[1] In lack of a better term we use "stack configuration" to refer to both the composition and configuration of the stack.

a full belly – results in little experimentation being performed. The weighing between exploration and exploitation is further influenced through dynamic re-parametrisation of the evolution logic: When the fitness of the best known stack is high, the parameter set is changed such that extreme modifications of the stack configuration become less likely.

Evolutionary approaches are infamous for often failing to reliably reach the optimum, but we do not expect that a truly optimal stack configuration can realistically be found through on-line experimentation in a short time frame, as neither the traffic or network environment conditions, nor the user or application requirements can be expected to be known at design time, and the search space is likely too large to exhaustively[3] explore. In fact, we do not even expect any of these to remain constant, as even the dimensionality and shape of the search space changes whenever a new protocol module is added to or removed from the system. Thus the goal of the optimisation process is to evolve a stack configuration which exhibits better performance than a generic stack. An example of such a generic stack is the Internet stack, which performs well under multiple traffic and networks conditions, but which is not explicitly optimised for the current network and traffic conditions. Naturally, we intend for the system's fitness to reasonably quickly approach the optimum, but not to actually reach it. We also do not assume that the resulting stack could be adequate for a wide spectrum of network conditions, but rather to be specialised for the situation under which it evolved.

Our network stacks consist of modular components or micro-protocols, that can be freely composed and configured within the limitations defined by a specification ontology we designed for this purpose. Network functionality – for example, protocols and services – is thus interchangeable at runtime. The stack configuration depends solely on the runtime-provided ontological specification. As the fitness specification is also runtime-defined, the optimisation process is inherently autonomous and does not require intrinsic (design-time) knowledge of the stack's components, functionality, the network layout, traffic conditions, or even the optimisation goal. The optimisation process is thus fully on-line-configurable and adaptable to the current needs of the users.

We wish to guarantee that experimentation is non-disruptive and – ideally – almost unnoticeable for the users and applications running on the system, i.e. induced changes in the operation behaviour need be small enough to be perceivable as fluctuations in the network conditions, etc. Therefore we employ an experimentation strategy that tries to schedule experiments such

[3] As most properties that define the search space and fitness distribution are specified at run-time, analytical approaches to reduce the search space are not easily applicable.

that possibly negative effects on the overall network performance are minimal. As mentioned before, our system adapts its own optimisation strategy depending on the perceived stack performance. If the users and applications are satisfied, i.e. if the performance is high enough, the effort spent on experimentation and adaptation is kept to a minimum, whereas low performance obviously warrants spending more time on exploration, and even risking temporarily lower performance.

Our concept for handling differences in the network environment was also inspired by nature. For example, finch populations evolved into a different species once they were isolated on the Galapagos islands, as they had to adapt to the environmental conditions and food sources there. We therefore venture that if the "living" – or network – conditions differ, the adaptation goal and therefore the optimal genome of the species – the stack configuration – will likely also be different. Our stack composition system therefore keeps multiple pools – called populations – which are conceptionally identical to the aforementioned isolated islands. Our system actively and passively measures and analyses the conditions in the network, and classifies the current situation. The baseline stack employed during the exploitation phase, as well as the candidate stack configurations used for exploration, are chosen from the population which was evolved under conditions that most closely resemble the current ones. Since we use on-line experimentation to rate each stack's performance, we also need to ensure that the conditions during the trials are comparable. While we use the same fitness function for all situations, the fitness landscape[ほ] can differ differ, because the utility of the stack is usually influenced by the network and traffic conditions. The classification and population selection process also helps to solve this problem, as it be used to guarantee that only stacks trialled under sufficiently similar conditions are compared with each other.

The classification and population selection process provides our system with a situational memory which is a vital means for recognising recurring network conditions and work loads: Consider a mobile device which is routinely employed in a wired, nearly loss-free and high-throughput environment, as well as in wireless networks that provide a lossy, low-throughput and high-delay service. The optimal stack configuration in the former case might utilise unmodified TCP, which is known to perform rather poorly in multi-hop wireless networks.[256] If the system can reliably and autonomously identify and discern between those conditions, that is notice the corresponding performance difference of the trialled stacks and assign them to different populations, we reason that each of these populations will over time evolve

[ほ] The distribution of fitness across the configuration space.

a different set of stack configurations, which are optimised for the corresponding situation in the network. When the aforementioned mobile device moves from the wired to the wireless network, the system can instantly switch to the most appropriate stack configuration, instead of slowly and gradually evolving towards a new goal state over and over again.

The stack modification process is designed such as not to impede the normal stack operations, i.e. when a new stack is created and the previous one is replaced, the ongoing communication sessions are not interrupted. This implies that state information must be persistently kept, understood and shared by all stack modules that provide the same service across stack recomposition. Consequently, the system must also be able to recognise faulty stack compositions and execution errors at run-time and to abort these with minimal delay and without loss of information.

Since we wish to support heterogeneous networks, in which the participating nodes can independently define their stacks, we need to ensure the capability to communicate across differently composed stacks. We define that the initiator of a communication flow, e.g. the client that connects to a server using TCP, defines the types and the order of the protocols that are used for communication.[⊏] Our stacks therefore always include all stack functionality, i.e. all possible protocols are made available for incoming connections. We explicitly allow for changes of the composition along the path – for example, encapsulation or tunnelling – as long as this happens transparently, i.e. the initiator does not become aware of it.

Lastly, as networks do not consist of one singular node, the optimisation process could not be considered useful if it was limited to optimising the performance of one node, but while doing so negatively impede the performance of the network as whole. We therefore designed the framework such that it can take the communicate and collaborate with other nodes within the network, and exchange measurement data and fitness information with them. This information can then be use into the fitness calculation, e.g. to achieve collaborative optimisation towards a common goal.

3.3 Stack Composition System

In this section we introduce the architecture of the stack composition system and describe its components. Our design is based on the autonomic control loop introduced in *Figure 2.4*. It encompasses the functionality of the network stack, the mechanics for modifying this stack, as well as the

[⊏] Nodes along the path are free to modify this order, in a similar manner as possible today for tunnelling or VPN, as long as the communication endpoints are unaffected by these changes.

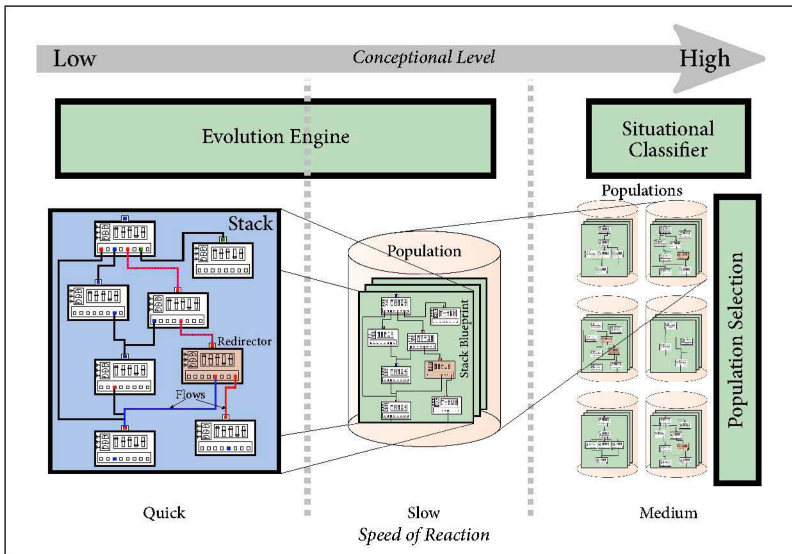


Figure 3.1: *A conceptional overview of the layers of adaptation within our stack composition system. The evolution engine gradually evolves the stack blueprints towards the optimum. The population selection mechanism operates on a pool of of stack blueprints, but with the help of the situational classifier can react to changes much faster than the evolution process. For even quicker and finer-granular decisions, the evolution engine can include in-stack redirectors in the stack, which operate on e.g. per-flow or per-packet basis.*

mechanisms for autonomously performing experiments and measurements, gathering and analysing information about the stack and overall system performance, and the logic for deciding how to adapt the stack. We begin with an overview of the framework architecture, followed by a detailed description of the components.

3.3.1 Layered Evolution – Long-term Stack Evolution vs. Mid-/Short-Term Adaptation

Based on the aforementioned considerations, we devised a design which includes three layers of adaptation and optimisation mechanisms, each utilising a different approach and complementing each other, as shown in *Figure 3.1*.

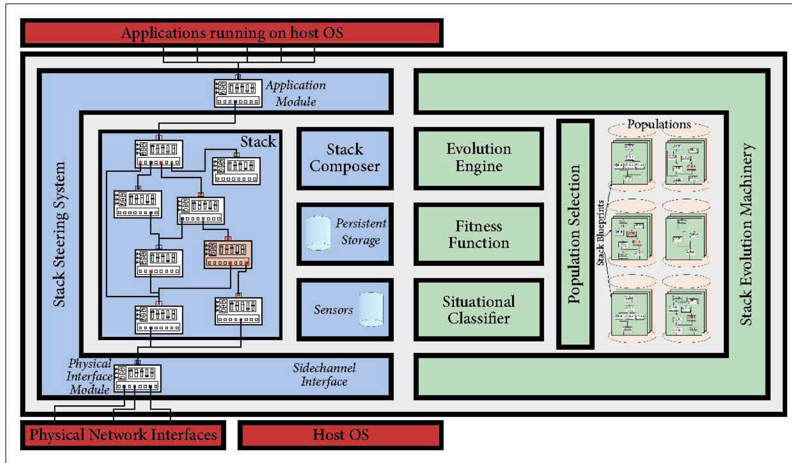


Figure 3.2: *The components of the stack composition system. Stack operations encompasses the stack steering system, composer, persistent storage space, and the sensors. The evolution machinery consists of the situational classifier, evolution engine, fitness function, and multiple populations of stack blueprints.*

The evolution engine constitutes the core of the adaptation functionality. It operates on a population of stack blueprints. A stack blueprint encodes the stack configuration, i.e. the connections between module instances and thus their interaction, as well as the configuration of these instances and thus their operational characteristics. The populations also store information pertaining to the experiments performed for each of the stack blueprints. The population evolves through repeated cycles of on-line experimentation and consequent application of an evolution logic, which in each successive cycle creates a new generation of stack blueprints based on the experimentally gained data and the fitness estimate derived thereof. Due to our requirement that the available micro-protocols, the definition of fitness, and the conditions in the network are unknown at design-time, we employ evolutionary algorithms and other machine-learning techniques that do not require design-time knowledge about the configuration space or the application environment. Consequently, the adaptation process is comparatively slow, similarly to evolution in nature.

Since changes in the environment can occur at a higher speed than the evolution engine can adapt at, we provide another layer of adaptation: The situational classifier analyses the state of the network environment, as well

as the ongoing traffic, and the population selector keeps a set of populations, between which it selects depending on this classification, which is based either on administrator-defined heuristics or unsupervised learning. Both the stacks used for normal operations and the candidate stacks used for experimentation are chosen from the population that most closely resembles the current situation in the network.

The third and fastest layer consists of in-stack redirector modules that are placed within the stack by the evolution engine. Redirectors are connected to two or more separate sub-stacks and decide on a per-call, per-flow, or per-packet basis to which of these sub-stacks the processing should be forwarded. Redirectors thus offer the finest granularity of adaptation in our system.

3.3.2 Framework Layout

The structure of our framework, which we call the stack composition system, is shown in *Figure 3.2*, were the components that encompass the stack mechanics are highlighted in blue, and those of the evolution machinery in green. *Figure 3.3* illustrates the interaction between these components.

The core networking operations are provided by the stack which consists of service module – a.k.a. micro-protocol – instances and a persistent storage space for keeping module state data consistent even across stack recompositions. The stack steering system manages and operates the stack, schedules experiments, and gathers performance measurements. The stack is constructed and modified by the stack composer, according to a stack blueprint selected by the stack steering system.

The evolution machinery encompasses and controls the three adaptation layers introduced above. The evolution engine produces a new generation of stack blueprints based on the parent generation's fitness as assessed by the fitness function. The situational classifier classifies the current situation based on sensory information about the state of the network and the traffic characteristics. The population selector maintains multiple populations and selects one of these based on the situational classifier's verdict and redirects all access by the evolution engine and the stack steering system to this population.

3.4 Stack Layout & Specifications

We require that the functionality of the network stack is made available as a set of modules, each of which encompasses a limited piece of core functionality and defines its capabilities and requirements for interaction with

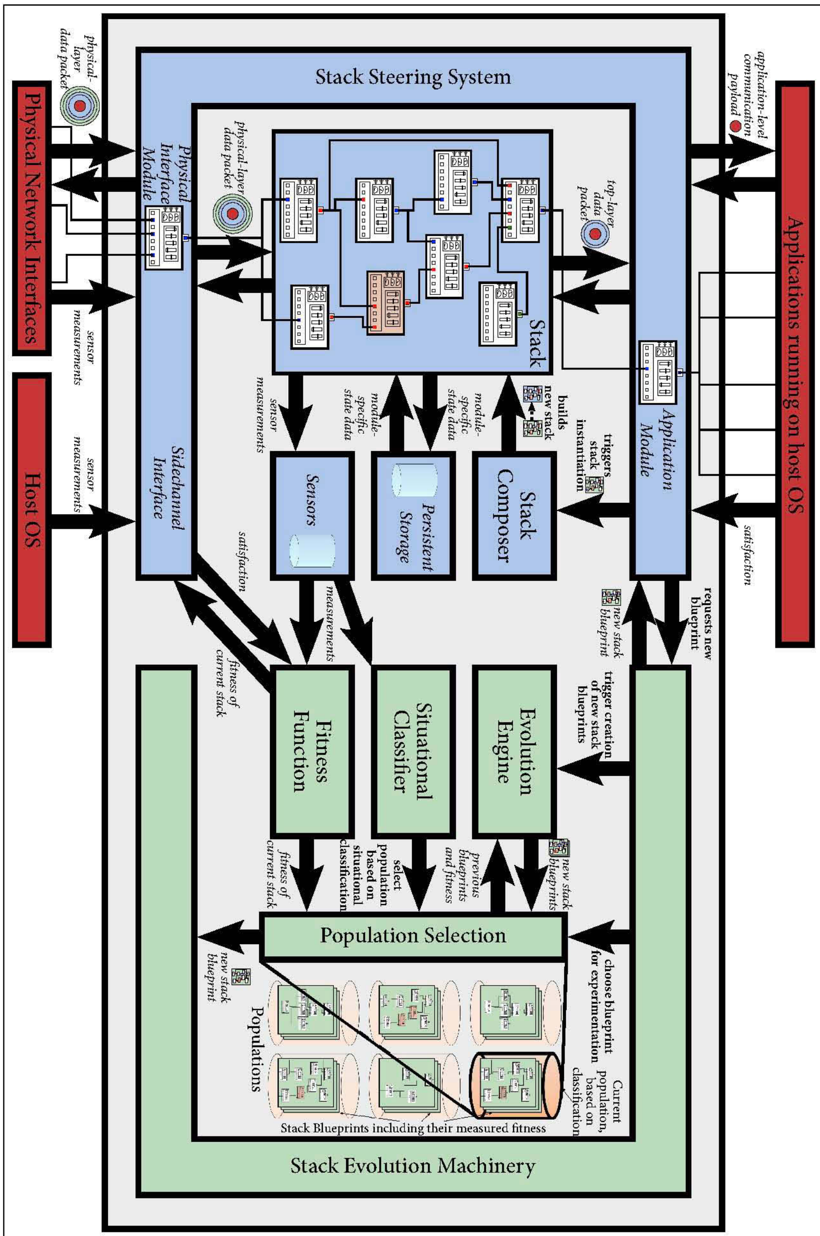


Figure 3.3: *The interaction of the components of the stack composition system.*

other modules through a fixed ontology. To guarantee maximum flexibility of composition, these modules are supposed to be co-operative, i.e. similar functionality should – whenever possible – be provided through identical interfaces which are generic to the functionality class. For further requirements for the protocol design, please refer to *Appendix A*. In this section we describe how modules can define and offer their functionality to other modules, specify their requirements for operation and interaction, and detail the means through which these modules can communicate, exchange data, control the program flow, and store state information.

Whereas our design shares many similarities and was inspired by other frameworks for stack or service composition such as Click,[197] RNA,[355] ANA,[38] etc., it is probably most closely related to SILO[96] and Da CaPo[282]: Like SILO, we also utilise micro-protocols, enforce constraints on them, and define configuration options that can be adapted, as discussed in *Section 3.4.1*. In particular, our design allows loops to occur within the call graph and only employs run-time checks to guarantee that the call graph is finite, as discussed in *Section 5.3.2*. And whereas SILO utilises the eponymous application-dependent silos which are dynamically generated per task, our systems constructs only one shared stack for all tasks, which can embed an intrinsic selection logic to branch between sub-stacks. Our in-stack redirectors and decision modules, which we introduce in *Section 3.7.6*, thus allow for recursive branching and decision making based on e.g. application-, flow-, or packet-type, within the stack. The embedding of the logic itself is subject to the stack evolution process. Similar to Da CaPo, our micro-protocol modules define their properties and constraints through a specification language. In our case, however, this property specification does not include the protocol behaviour, only its requirements for composition and possibilities for configuration. Instead we determine the effects that protocol composition and configuration has on the utility experimentally, based on the actual conditions in the network at the time of deployment.

We designed a communication interface that provides the functionality needed to support most networking protocols and which we describe in *Section 3.7.3*. This interface is conceptionally similar to the x-kernel's interface,[164] but also includes name resolution facilities that are related to to the approach of Plutarch's.[72] But module interaction is not limited to this interface, as our fundamental design for (micro-)module interaction differs significantly from most other stack composition architectures and is closer related to the approaches employed in software engineering, e.g. by Alagar *et al.*,[10] as it resembles the functional interfaces exposed by objects in programming languages like C++. Execution does not linearly traverse a stack of modules downwards, but akin to function calls returns to the call origin,

enabling the use of common programming constructs like loops, branches, etc.

3.4.1 Stack Modules

The network stack is built out of modules which separate the protocol and service functionality into small and potentially re-usable blocks. The modularisation of protocols into atomic stack building blocks, which are also known as micro-protocols[267] as introduced in *Section 2.7.1*, adds flexibility to the stack composition process: It enables not only the re-ordering, but also the re-use of functionality at different places within the stack – provided that conceptionally compatible functionality is exposed through a common functional interface.

For the purpose of stack composition, modules are defined by

- the service they provide and a corresponding service identifier,
- the way they can connect to and communicate with other modules,
- the control parameters they expose, and
- the sensor measurements they provide.

The service identifies the type of functionality offered by the module, and compatible services are supposed to use the same service identifier. This unique identifier further defines the calling interface used for communication between module instances. We further elaborate on this design and the reasoning that guided it in *Section 3.7.2*.

Modules can expose control parameters that influence operational characteristics of their service, for example, the type of and number of bits used for error correcting codes in a module that provides forward error correction. The module specification dictates the range of values these parameters can be assigned by the evolution engine and subsequently encoded in the corresponding stack blueprint.

Modules can utilise the functionality provided by other modules through the definition of connectors, which can be imagined as smart, strongly-typed pointers. For example, a communication protocol can define a connector with the service identifier of the above-mentioned error correction facilities, and be assured that the calling interface conforms to the interface specification. However, which error correction algorithm, i.e. which module, is going to handle these calls, is not known to the caller, as the association is set up and maintained independently by the stack composer in accordance to the stack blueprint created by the evolution engine.

Modules can also provide sensor data to other modules. Sensors allow read access to internal state or measurement data, for example the current network load or communication error rate, and are introduced in *Section 5.1*. Apart from the inherent benefits offered by cross-layer information sharing, this data can also be used by the in-stack redirectors (see *Section 4.3.1*), the fitness function (see *Section 5.3.5*) and the situational classifier (see *Section 4.2*).

Module features are defined by means of a formal specification language. A module specification has to accompany every module and is parsed at run-time whenever a new module is loaded into the stack composition system. This specification is fully defined the context-free grammar presented in EBNF form below, and constrains and guides the operation of the evolution engine and the composer.

```
module = "{", service-id, { ",", required-feature-id-list }, ";",
         {requested-feature-id-list} ";" { control-list },
         ";", { connector-list }, ";", { sensor-list }, ";",
         trial-time, "}" ;
required-feature-id-list = feature-id-list ;
requested-feature-id-list = feature-id-list ;
feature-id-list = feature-id, { ",", feature-id } ;
control-list = control, { ",", control } ;
connector-list = connector, { ",", connector } ;
feature-id = identifier ;
control = identifier, "=", intrange, ",", weight ;
connector = identifier, ":", service-id, { ",", feature-id-list }, ";",
            weight, ";" ;
sensor = identifier, "=", intrange;
trial-time = digits ;
identifier = alpha, { alnum } ;
intrange = "[", integer, ",", integer, "]" ;
weight = "0.", digits | "1.0" ;
integer = [ "+" | "-" ], digits ;
alnum = alpha | digit ;
digits = digit, { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
alpha = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
        "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
        "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

The *service* is denoted by a globally-unique and standardised service identifier, which explicitly defines the calling interface. Additional service *features* may be specified, which further detail the provided functionality, but do not change the interface. *Control* specifications consist of the integer range that defines the domain of possible control values, and a boolean stating whether the range is nominal or continuous. *Sensor* specifications

also consist of an integer range of possible values the sensor may report, and a sensor identifier which is used by other entities to access the provided information. *Connectors* in turn consist of a target service identifiers combined with an arbitrary number of required and/or preferred features. The *trial-time* specifies the minimum time needed for experimentation with this module, as detailed in *Section 5.3.2*.

The specification language enables the system to specify the properties of the provided protocol or service, as well as its requirements for services offered by other modules. This approach shares some similarities with e.g. the L[281] language used in Da CaPo,[282] in that it constrains the possible compositions of the stack. In our case the criterion for optimality is however not encoded within this specification, but based on experimentally gained fitness estimates. The constraints defined by means of this specification do not explicitly impose an order of precedence as e.g. is the case for SILO,[366] but implicitly by enforcing a compatible service interface for all connectors.

3.4.2 Stack Composition

The stack composer provides the mechanics for stack composition, i.e. it handles module instantiation and destruction, initialises the module control parameters, sets up the connections between the modules, and provides access to the persistent storage space. The composer constructs the network stack based on an abstract composition blueprint provided by the evolution engine.

Stack Blueprint

This composition blueprint specifies the entire stack composition, including how many instances of each module are present in the stack, how they are configured, and the connection between these instances. Every available stack module class is instantiated within the stack blueprint at least once, to ensure that incoming data can be reliably de-multiplexed and forwarded to a module instance that is able to handle it, as discussed in *Section 3.7.5*. The blueprint further specifies how to connect these modules to the persistent modules provided by the stack steering system which provide the link between the stack and the outside. Every blueprint generated by the evolution engine is guaranteed to be valid, in as much as it is statically checked to conform to the specifications and fulfil the assumptions made in this section. These stacks may nevertheless be inoperable, as e.g. the halting problem is generally undecidable for Turing machines.[362] And since not every cycle in

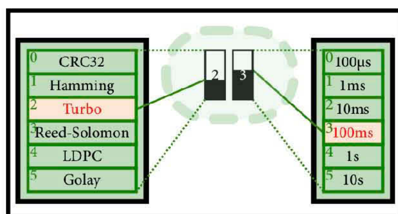


Figure 3.4: *An example of how control values map to protocol settings. The value 2 of the left control selects Turbo Codes for error correction, whereas the value 3 of the control on the right corresponds to a time-out of 100 ms.*

the stack graph necessary leads to an infinite loop,[ほ] we decided not to assert that the stacks are loop-free. The stack composition system consequently also provides mechanisms run-time validation, which are described in *Section 5.3.2*.

Controls

As stated before, control parameters specify the internal configuration of modules. Which configuration details are exposed, and how they affect the operation of the module is left to the module developer. As a guideline, all settings for which manipulation by the stack composition system might have a beneficial effect on performance should be exposed as controls. For example, a module implementing Reed-Solomon codes might provide a control to set the number of symbols to use, as the correct setting depends on the network conditions, i.e. the likelihood of transmission errors. The internal mapping of the control range to actual module parameters is again left to the implementer. For example, the range [0, 2] could actually refer to time-out delays of 1, 10, or 100 ms, respectively.

Controls can be designated as either nominal or quantitative, as shown in *Figure 3.4*, which influences the operation of the evolution logic. For nominal controls, there is no inherent relationship between neighbouring values, as e.g. whether a control that selects the forward error correction method maps Hamming, Reed-Solomon, BCH, Golay or turbo codes to 0, 1, 2, 3, 4 or any other permutation thereof is unimportant. Thus the evolution logic can modify such controls at random. For quantitative controls however, neighbouring values are more closely related than distant ones, for exam-

[ほ] Limited re-evocation of the same protocol is possible in the current Internet stack as well, e.g. for tunnelling of IP or Ethernet packets over a VPN service that runs on top of UDP.

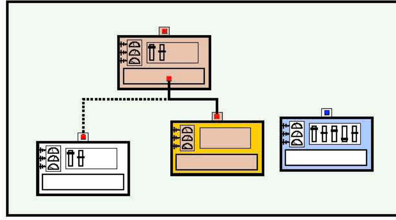


Figure 3.5: *An example of how connectors wire module instances together. The single connector of the module on the top can connect to one of two instances. The solid line indicates the instance it points to, whereas the dotted line indicates a candidate instance. The service identifier of the module on the right, indicated by the coloured squares, is incompatible (blue instead of red), thus no connection is possible. Connectors may also not point back to the instance they are located in.*

ple, changing the number of Reed-Solomon symbols to use from 7 to 8 has a lower impact than a change from 7 to 13. The distinction between nominal and quantitative controls thus allows the evolution engine to operate more efficiently, by for example non-deterministically deriving the value of a control using a Gauss distribution centred at the previous value, as we discuss in *Section 4.1.3*.

Controls have associated weights which specifies their importance for the module's operations, i.e. state how much a modification of the value is expected to alter module operations and thus performance. The weight is needed by some of our evolution logics, as they require a distance metric for stack configurations.

Connectors

As further discussed in *Section 3.7.2*, modules can access the functional interfaces of other modules through connectors, i.e. dynamically-typed pointers. Just like the type concept in programming languages guarantees that the result of an expression cannot be assigned to a variable of incompatible type, connectors can only point to module instances that provide the service which the connector requires, as illustrated by *Figure 3.5*. The module instance the connectors point to is encoded in the stack blueprint. The actual connection is set up and maintained by the composer. The stack blueprint is guaranteed to honour all requirements encoded in the module specification, thus connectors are only bound to module instances that offer the requested

service type and provide all required features. Connectors may not be left unconnected either, thus null pointers do not occur. Consequently, missing services or required features are treated as a fatal errors which prevent the instantiation of a stack. Apart from the required features, the connector specification may include requested features, which should be fulfilled whenever possible, but which the evolution engine may decide to ignore at will. Connectors also have an associated weight which specifies their importance to the module performance, as already discussed for controls above.

Compositional Control of Processing

The composition system is able to influence the processing flow in two complementary ways, by either setting the value of a control parameter, or by selecting a different module instance for a connector to point to. Additionally, it is free to instantiate the same module multiple times, and configure and connect each of these instances independently. The system is further able to place in-stack redirectors at arbitrary positions, as discussed in *Section 4.3.1*.

Module implementers can freely choose how to expose configuration options to the system, e.g. whether to provide different modules for Hamming codes and for Reed-Solomon, or instead combine the functionality of both algorithms in one module and define a control to select between them. From the compositional point-of-view both approaches are identical.

3.5 Stack Mechanics

In this section we describe three of the four components that provide the stack mechanics, i.e. the stack steering system, the composer, and the persistent storage. The last component, the sensors, are discussed in *Section 5.1*.

3.5.1 Stack Steering System

The stack steering system links the stack composition system with the outside. Conceptionally, the stack steering system replaces the operating system's network stack, and dynamically links its internal modular stack to the applications at the top and the network interfaces at the bottom. It directs the communication flow between the network stack and the outside, by channelling incoming packets to the module instances that constitute the stack and forwarding outgoing data to the applications and onto the network. It schedules stack operations, and signals network and timer events to the appropriate modules, and gathers and monitors measurement data from inside

and outside the system. The stack steering system also monitors the operations and call flow within the stack and thus detects and handles execution error, e.g. exceptions or segmentation faults, and composition errors such as infinite loops. The stack steering system further controls when the current stack is replaced with a new one, either because of changes in the network and traffic conditions as we discuss in *Section 4.2*, or to experimentally evaluate the candidate stack blueprints that the evolution engine provided as we describe in *Section 5.3.2*. The interaction with the outside and system-specific implementation details are presented in *Appendix B*, while *Chapter 5* describes the process of gathering sensor data and assessing the information contained therein, as well as the interaction with other entities in the network.

3.5.2 Stack Composer

The composer provides the stack composition mechanics, i.e. it handles module instantiation and destruction, initialises the module control parameters, sets up the connections between the modules, and provides access to the protocol state storage. The composer constructs the network stack from the abstract stack blueprint, which we discuss in *Section 3.4.2*, and is mostly responsible for management tasks, such as resource allocation and reuse thereof. It implements an instance "garbage collector", which decides whether to re-configure existing module instances, keep them around in a dormant state for later re-use, or to destroy them and reclaim the allocated resources. It signals module instances about impending changes, i.e. before they are started, stopped, or destroyed, so that they can, for example, deal with on-going connections with remote nodes as appropriate.

3.5.3 Persistent Storage

Similarly to DRUID,[354] the module instances in our stack composition system can keep persistent state. The persistent storage facility provides instances with the means to store arbitrary information and to share data amongst each other. The data stored in this facility can be either unique to a module instance, shared among instances of the same module, among instances that provide the same functional interface, or globally accessible to all instances running within the same node. Data access is based on keys and protected by a simple access control methodology. All data apart from the instant-specific type is persistent, i.e. it does not change when the stack is reconfigured. This feature enables the stack composition system to, for example, exchange communication protocol implementations at run-time with-

out losing connection information, if the protocol implementation keeps the corresponding information in the persistent storage space. Our implementations of of TCP, DCCP, and UDP, for example, utilise the same shared data structure to maintain state information about ongoing communications, and thus an existing TCP connection can be taken over by UDP when the stack is recomposed and the pending data in the send queue forwarded – but of course without guaranteed sequentiality and unreliably – as discussed in *Section 3.7.2.*

3.6 Evolution Machinery

The evolution machinery constitutes the "brain" of the composition framework. Its components and their interaction are depicted in *Figure 3.6.* It contains the functionality for inventing and selecting between the stack compositions, by providing the stack blueprints which define the stack configuration, i.e. the configuration of the module instances that build the stack and the interactions between them. It further houses the situational awareness and related adaptation facilities.

The evolution machinery consists of the evolution engine, which generates stack blueprints based on the experimentally measured fitness of the previous generation of stack configurations, the situational classifier, which assesses the current situation within the network and decides which class it resembles most closely, the population selector which selects a population of blueprints based on this classification, the fitness function, which determines how well the tested stack is suited for the current situation and which is described in *Section 5.3.5*, and lastly the in-stack redirector modules, that are encoded within the stack blueprint and which conditionally adapt stack operations based on fine-granular decisions. Whereas the algorithms governing the evolution are explained in *Chapter 4*, the integration of and interaction between the machinery's components are detailed in here.

Evolution Engine

The evolution engine generates stack blueprints, i.e. plans describing how an instance of the stack is to be composed and configured, which we discuss in *Section 3.4.2.* The evolution engine contains the functionality for inventing new and selecting between stack compositions, and provides the stack steering system with a set of stack blueprints. For this purpose it applies an evolution logic, i.e. an optimisation method which generates a new generation of candidate stack blueprints using machine learning techniques, as explained in *Section 4.1.*

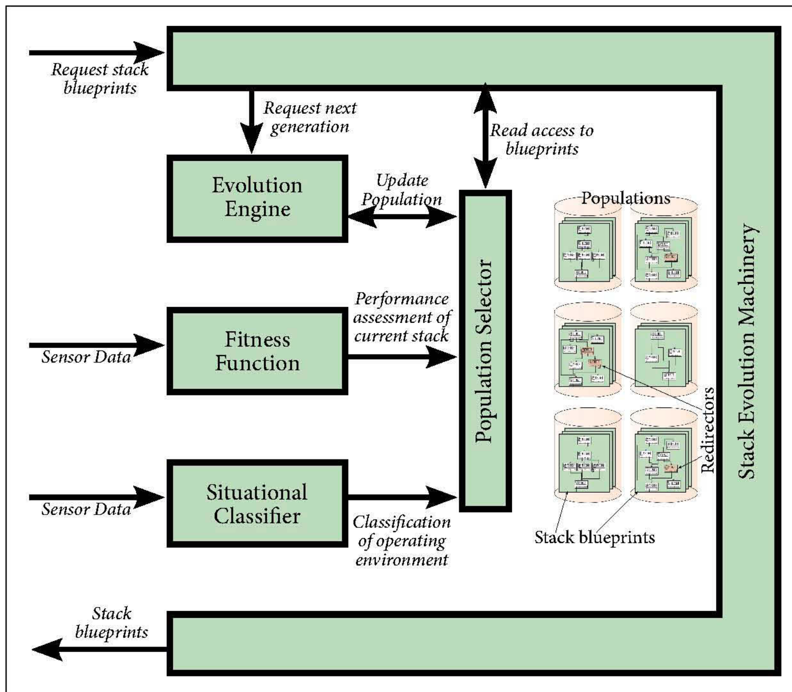


Figure 3.6: *Overview of the components of the evolution machinery and their interaction. The in-stack redirectors modules are shown as dark brown boxes inside the stack blueprints stored within the populations.*

The selection of evolution logic used for generating the stack blueprints, as well as their configuration, can be either statically configured or dynamically adapted at runtime. In the latter case the fitness of the best composition found so far guides the decision, as described in detail in *Section 4.1.4*.

Whereas not required, background knowledge about the behaviour and dynamics of the functionality provided by the stack modules can be used to speed up the adaptation process by limiting the number of possible stack configurations that can be explored. For example, while the in-stack redirectors we introduce in *Section 4.3.1* are very versatile, their use also increases the size of the configuration search space. The system configuration therefore allows to place constraints on the minimum and maximum amount of instances for each module class individually, as well as on their configuration.

Population Selector & Situational Classifier

The situational classifier categorises the current situation of the system based on sensor measurement data gathered from the running system, i.e. it reports the same class identifier for all measurements it considers sufficiently similar. The population selector maintains a set of populations and uses the classifier's output to select between them. It controls all access by both the evolution engine and the stack steering system to the populations and thus guarantees that they both operate on the population that the situational classifier assessed as most appropriate. Thus the deployed network stack is always chosen from the population that evolved under the most similar conditions, both when exploring new candidate stack configurations, and when exploiting the best stack found so far. The logic which controls the classifier can be either an administrator-defined heuristic or a unsupervised learning method, as described in detail in *Section 4.2.* The classification method and the criteria based on which it operates can be configured at run-time. For the future we investigate the possibility of letting the system determine adequate criteria on its own.

3.7 Handling Modular Stacks

As stated in *Section 3.4.1*, the network stack is composed of multiple, collaborative module instances, which can be replaced without interrupting communications. In this section we describe how our system guarantees smooth network operations across such changes.

3.7.1 Module Instance Life-cycle

The code for all available modules is kept in memory throughout the entire operational period of the stack composition system, as one instance of every module is required in every stack instance. Similar to object instances in languages like C++, a module instance consists of a private memory area in which the specific local configuration data is kept, i.e. the settings of all controls and connectors, as well as local variables internal to the module.

The composer manages a pool of instances. Whenever a new stack configuration is realised, the composer either selects an instance from this pool or creates a new one as required. In addition to the normal object constructors and destructors, modules provide `init/exit` functions which are invoked whenever an instance is added or removed from a stack, and which enable the modules to adapt their internal set-up depending on the control settings. At this point in time, the invocation of connectors and thus communication

are not possible. Modules also provide start/stop functions, which are executed immediately after the stack has been set-up and before it is disabled. At the time when these functions are called communication and calls to connectors are possible, thus messages can be exchanged between modules and remote entities, and timers initialised.

Modules can be added to and removed from the system by external programs at run-time. In our current design these messages are transmitted system-locally by means of Unix-domain sockets, and thus protected by the operating system-imposed access restrictions. We intentionally externalised the code-deployment process from our system design for practical reasons, i.e. to be able to leverage the operating system's services. We assume that an external application, which runs on top of the network stack is responsible for mobile code deployment, e.g. using one of the technologies introduced in *Section 2.7.3*, and informs the system whenever a new module is available. Our API for this purpose is very simple and consists of only two functions. load adds a new module to the system, and takes two parameters that define the location of the dynamic library that contains the module code and the text file that contains the module specification, respectively. The unload function in turn removes a module from the module pool and again takes the path to the dynamic library as parameter. Since every available module is instantiated at least once within every stack, the addition or removal of modules invalidates the experimentally gathered fitness estimates. We therefore re-start the adaptation process, but do not begin from scratch. Whenever a new modules is added, the stack configurations are modified to include an instance of the module, but the controls and connectors are not modified. This new instance is therefore initially unconnected, and subject to the adaptation regime imposed by the evolution logic, as we intend to bias towards gradual introduction of new features. If a module is removed from the pool, this technique is not applicable, thus we discard all stack configurations in which connectors point to the affected instance(s) and replace them with randomly initialised ones. The experimentation process is then restarted.

3.7.2 Interchangeable Modules

Knowledge about the type of service that a module provides is a vital prerequisite for efficient runtime service or stack composition. Our system utilises a simple ontology to exclude connections between obviously incompatible modules: Similar services are supposed to utilise the same interface, which is identified by a unique service identifier, similar to the type-of-service fields of Ethernet or IPv4, or the known ports of TCP and UDP.

Even though, for example, different web servers offer a wide variety of features and services, the calling conventions are standardised by means of the HTTP. Similarly modules which provide a compatible service in our design expose the same service identifier, which guarantees provision of and conformance to a service-specific functional interface. Whereas one generic interface for inter-connecting modules can be used for communication purposes,[38,355] we decided to follow the example of object-oriented programming languages, and modelled our communication paradigm after the interface classes in C++. Instead of limiting the interaction between modules to a single interface – which may or may not be sufficient for future needs – we thus leave the definition of the interface to the service providers.

For explanation purposes, consider a module which provides forward error correction, a vital service in noisy communication environments. Here the interface could consist of two functions, `encode` and `decode`, which both take a variably-sized byte buffer as input and return another one on success or `null` on error. Calling modules cannot directly select which implementation, i.e. module instance, to use, only which functional interface to access. Which module instance to use is decided by the evolution logic instead.

The previous example also illustrates another feature of our design: Modules are not chained together as in many other architecture that offer composition facilities,[39,187] instead the control flow returns to the caller: As opposed to the sink/source-model, our modules do not take input data from one module, process it and produce output data that is forwarded in sequence to the next module. Our design thus follows the paradigm of object-oriented imperative programming languages: Modules can invoke functional interfaces at any point in their code control flow by means of connectors. For example, assume a communication protocol module which requires error detection functionality. Instead of implementing code that calculates a checksum and appends it to the message, the developer defines a connector `con_ecc` with the associated service type identifier `forward error correction`. The module code may now call this connector in the same manner it would normally access an object in C++, i.e. use

```
encoded_payload = con_ecc->encode(payload);
```

to invoke the error correction module for encoding the original message `payload`. As control is returned to the calling module once the encoding is complete, the communication protocol may then perform further processing as needed, e.g. update the header fields to reflect the length of the encoded payload and put the packet onto the outgoing communication stream, or call the connector of a module that offers compression before adding er-

ror correction codes. Thus implementers can separate protocol functionality into atomic components in the same way they do when separating it into multiple libraries or classes. For simplicity reasons the program flow of our current implementation is synchronous and single-threaded, but utilises queues and provides signalling and timer support. The design itself supports asynchronous operations and multiple threads, which we consider implementing in the future.

The service type that is part of a connector's specification guarantees that only module instances that provide the matching interface can be assigned to a connector. The calling module can thus be certain that the instance pointed to by the connector implements the required functionality according to the interface specification, and also that it offers all required features. Interface design is left to the implementers and can be as specific or generic as required, and we do not impose any calling conventions, protocol header formats, naming schemes, etc. The Unified Communication Interface interface which we describe below, for example, is sufficiently generic to handle such heterogeneous protocols as TCP, IPv4, or Ethernet, but other schemes could be realised within our system just as easily. All these protocols provide sufficiently similar functionality, they forward information between endpoints, identified by source and destination addresses. Requirements for a specific subset of the functionality, e.g. protocols operating in the Internet transport layer (TCP,UDP, etc.), can be specified through the definition of the required features in the connector specification. Similarly generic interfaces haven shown to be versatile enough for most communication tasks[72,164,183] and increase the possibilities for stack composition, as for example shown in *Figure 3.9* below.

As introduced in *Section 3.5.3*, modules can share arbitrary persistent data structures amongst each other. These structures are identified by a key similar to way in which IPC shared memory access is handled. Since the data is kept around even if the creating instances are deleted (unless explicitly freed), and keys can be chosen freely by the implementation, this concept allows different instances of the same module, different modules that provide the same interface, and even seemingly unrelated modules to share information amongst themselves, even if they are not active – or not even instantiated – at the same time.

Practical Substitution Example

One example of shared protocol state – which we implemented and use – is connection state sharing between TCP and UDP. Whereas UDP itself is not connection-based, keeping this information in a shared structure en-

ables the system to switch from TCP to UDP and back without breaking the connection. Naturally, a switch between TCP and UDP should only be performed for traffic which does not require reliable or sequential transmission (see *Section 3.7.3* below), as UDP does not fulfil these requirements. Using TCP to transmit data in an unreliable and unordered way can be beneficial in some situations, e.g. to maintain TCP-friendliness. When switching from TCP to UDP, incoming data is assumed to directly follow the highest previously received TCP segment. UDP updates a shared counter recording the number of payload bytes received. When the system switches back to TCP, this counter is used in place of acknowledgements to validate the reception of data after the highest segment transmitted during the previous TCP operation. A similar process is in place to handle TCP timers which are restarted as appropriate, and connection terminations which happen when no UDP packets are received within a specific interval, etc.

3.7.3 Unified Communication Interface

Networking protocols are often implemented in a way the hinders composition or even makes it impossible, for example, because they depend on specific underlays. In here we therefore present our approach for stack components that are suitable for composition, yet compatible to existing standards, and the way these components interact with each other.

Motivation & Overview

Similarly to many other approaches[164,197,354] that utilise dynamically composed protocols and services, our system provides a unified interface for communication protocols which abstracts access to protocol functionality. The group of protocols which we – in lack of a better descriptor – denominate as "communication" protocols, encompasses protocols as diverse as TCP, IPv4, Ethernet, etc., i.e. all protocols that take a data payload and forward to a specified destination, no matter whether this destination is identified by a port, DNS name, IPv4 or Ethernet address. The need for such a generic interface for our system is obvious: To enable the stack composition system to freely exchange or re-order communication protocols, the higher and lower layer protocols have to be able to interact with each other. The common Internet protocol implementations, however, require intrinsic knowledge of e.g. the address format to forward packets to the lower layers. For example, assume that an application wishes to communicate using TCP, which in turn is connected to IPv4. Current designs require the application and the network stack to implement logic that inherently knows that TCP operates on

16-bit port numbers, and that IPv4 uses 32-bit addresses. Thus the application usually needs to pass the address in an underlay-dependent format – for example, using the correct `sockaddr struct` in POSIX – from which the protocol implementations then extract the protocol dependent part, i.e. port and IP address. This naturally poses problems when new protocols are added to the stack, for which reason `getnameinfo` and similar underlay-independent functionality was added as an afterthought to the POSIX specification when IPv6 was introduced. Thus applications can at least be implemented without requiring knowledge about the IP layer's address format or the size of the port field, but still have to provide both separately as host and service fields, and thus are limited to exactly two underlying protocols. The problem for the stack implementer however remains, as they have to know beforehand what protocols are available, what their addresses look like, how they are connected, etc., a luxury not available to our stack composition system, which allows protocols to be added at run-time.

We intend for the system to be oblivious of the protocols' intrinsic design and addressing formats, as we do not presume to know whether future protocols use ports, service names, host addresses, or something else which has not been invented yet. Neither do we assume to know how these protocols are most effectively combined. And obviously, we also do not want designers to infuse knowledge into their protocols about what lies below them, as this thwarts all efforts towards dynamic stack or service composition. Thus we designed the Unified Communication Interface, which abstracts the intrinsic format of addresses into a common and easily extensible format. Protocols and applications that use this interface cannot and do not need to know anything about the stack's actual composition, which ensures that arbitrary communication protocols can quickly be integrated into the stack composition system. Address resolution is also encompassed within and abstracted by this concept. As a plentitude of naming and addressing schemes has been proposed (see *Section 2.7.2*), we did not intend to offer yet another new scheme, but instead limited our efforts to simply "abstracting away" the incompatibilities between the current protocols, and provide an interface that is flexible enough to support almost any possible addressing and resolution scheme. And while we utilised the scheme we describe below for our protocol implementations, all other schemes that are compatible with the constraints imposed by our module specification language defined in *Section 3.4.1* – i.e. all schemes we are aware of – can be integrated into our system just as easily.

The functionality provided by common networking protocols such as IPv4 on their own is however not sufficient to support this interface, as it inherently requires address resolution support. We therefore decided to pro-

vide meta-protocols which – similar to the compartments of ANA[16] – combine address-resolution, forwarding, routing, etc., into one single virtual entity. Thus a "meta-ARP" module might not only glue together the necessary micro-protocol modules needed to implement ARP functionality, but also leverages an underlying protocol, e.g. Ethernet^, for packet forwarding.

Unified Communication Interface

The interface through which communication modules interact consists of only four functions:

register notifies the module of a new entity on top, which intends to utilise the communication facilities.

unregister does the opposite, i.e. is invoked when the entity becomes unavailable.

send forwards a data payload to another registered entity. The address of the destination entity is resolved internally by this method, if needed.

incoming is invoked whenever a packet arrives which is destined for this module instance. This callback function is required by the demultiplexing logic described in *Section 3.7.5*.

connect opens a channel for stream-oriented communication.

disconnect closes the same.

lookup resolves an address. The same functionality is also included in send above.

Address Resolution

Registration makes the entity available for incoming communication and can involve the publication of an address under which it is reachable within the realm of the protocol or meta-protocol. Our interface supports two distinct methodologies. We use *caller* to refer to the entity which invokes a connector, and *callee* for the instance the connector points to which provides the service the caller requested.

Callee-defined addresses are provided by the called entity. An IPv4 module could, for example, allocate and return a new zero-configuration IP address such as 169.254.12.123 (see RFC 3927[51]). An Ethernet module in turn

^ Or any other protocol the evolution engine decides on.

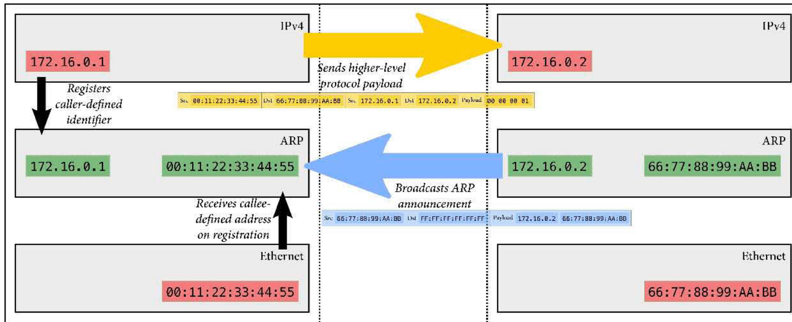


Figure 3.7: *This example illustrates the interaction between modules using both caller- and callee-defined addresses. Here the IPv4 module registers with the meta-ARP module using caller-defined addressing, while the meta-ARP module uses callee-defined addressing to interface with the underlying Ethernet-module. The packet encoding illustrates a subset of the data that is actually put on the wire.*

could return the concatenation of the EtherType field and the interface's MAC address.

Caller-defined addresses are provided by the calling entity and are handled by the name-resolution facilities of the module. For example, consider the case were IPv4 registers with the meta-ARP module which leverages Ethernet for communication. When the IPv4 module registers an IP address, the meta-ARP module stores this address in an internal table, and responds to incoming WHOHAS messages for this IP address with its own Ethernet address. Likewise, a meta-mDNS module, which sits on top of IPv4, registers the higher level name host1 as host1.local which it maps to a zero-configuration IPv4 address provided by the underlying IPv4 module.

When registering by means of this interface, the caller must specify a globally unique protocol identifier for compatibility purposes. This identifier is used e.g. by Ethernet to correctly map IPv4 to an EtherType of 0800h and by IPv4 to map TCP to 6 in the protocol field. Unknown protocols are mapped to dynamically allocated values in the experimental ranges.

The interaction between caller- and callee-defined addressing schemes is shown in *Figure 3.7*. Here a meta-ARP module links the underlying Ethernet protocol to the IPv4 above. As we require compatibility with existing protocol implementations, meta-ARP registers twice with the underlying Ethernet module, once using the protocol identifier for ARP and once using the identifier of the caller, i.e. IPv4.

The type of the address – single-cast, multi-cast, broadcast – is indicated by a flag or bit-mask supplied to the function. Protocols are free to only support a subset of these types, as indicated by the features defined in the module specification.

Traffic Class

Another feature of our interface is the type-of-service field traffic class, which consists of four bits encoding boolean values.

reliable This boolean specifies whether the payload is supposed to be reliably transmitted, i.e. using a protocol like TCP, which re-transmits lost or corrupted packets.

sequential This flag states whether sequential data transmission is required, i.e. whether packet re-ordering is acceptable or not.

priority These two bits denotes the priority class of this traffic, depending on how time-critical the delivery of the payload is.

This field is used for QoS provision, and determines e.g. the content of the differentiated service field header field in the IPv4 header (see RFC 2474[261]), or the underlay used by a protocol multiplexer. Real-time VoIP traffic would thus only set the priority level 3, but no other flags, as for this traffic type packet loss and re-ordering are preferable over higher delay. Bulk FTP transfers instead require reliable and sequential transmission, but are not time-critical, thus priority level 0.

Implementation

The communication protocol interface forwards payload data using a zero-copy memory buffer (for the bottom-up direction, i.e. incoming packets), and via an efficient double-ended vector structure (top-down, i.e. for outgoing packets).

For examples of our protocol modules which use this interface, as well as a discussion of the implementation details and necessary deviations from or extensions to the pertaining standards, please refer to the *Appendix B.2.1*.

3.7.4 Reliable Communication Channel

Since we intend to support collaboration between stack composition system instances running on remote nodes, we provide a reliable communication channel which is independent of the stack used for experimentation. Sensor

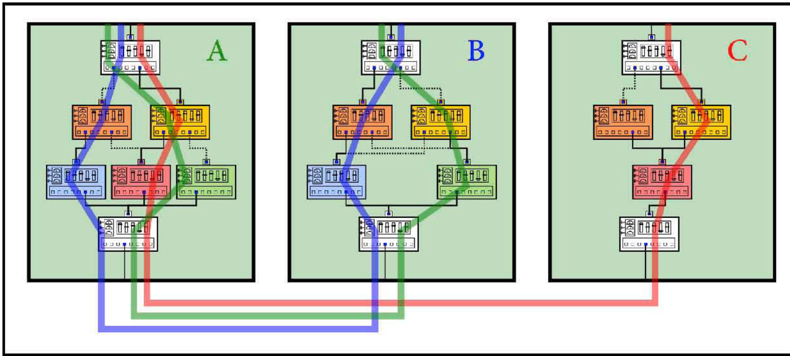


Figure 3.8: *The module execution flow depends on the sender's stack composition. Incoming data is forwarded based on the identifier encoded in the data stream and mirrors the sender's execution flow in reverse.*

data required for the evaluation of the current stacks fitness should, for example, not depend on an experimental – and potentially unreliable – stack, as this might distort or even invalidate the result, e.g. by delaying or inhibiting the effective transmission of necessary information from and to remote nodes[ヒ]. Alongside the trialled stack, a known-good and resilient stack composition is therefore active in our system, which can be either a generic (e.g. the Internet) stack or a stack configuration that has proven to operate reliably in the majority of application environments experienced so far. This stack is used to transmit stack composition system control and status information, such as sensor data, status reports, etc., between nodes. Whereas true reliability cannot be guaranteed this way, as for example a malfunctioning protocol could effectively separate the node from the network by flooding a network link, we aim to achieve a higher level of resilience through this provision of an independent side channel.

3.7.5 Sender-Initiator-Defined Composition

In our design the stack of the transmission's initiator defines the control flow between the module instances and thus the data processing order and packet composition for the entire flow. This naturally implies that the packet's recipient has to be able to traverse the same path through the stack in reverse

[ヒ] Adequate module design can help to reduce the likelihood of such an event, as we discuss in *Section A*

order to decode the packet. As the stack composition system on every network node is independently experimenting with and evolving its stack, the communication partners are likely to utilise different stack compositions. These compositions are obviously not static and change e.g. whenever a new stack is used for experimentation. The module instances in the stack are also further able to chose between different paths through the stack based on arbitrary criteria. In-stack redirectors, which we introduce in *Section 3.7.6*, for example, can be used to select a different sub-stack per flow.

This concept is illustrated by *Figure 3.8*. All three flows depicted there follow a different path through the stack, based on the stack composition of the node that initiated the flow. The green flow was initiated by the node A, and therefore follows the path defined by A's stack configuration. Likewise, the blue flow was initiated by node B, and the red one by node C. This figure also illustrates one problem that can occur when not all protocol modules were available on all nodes: Here nodes B and C cannot directly communicate, because they are missing a common protocol module. Such problems can be prevented by providing a common code base to all nodes – as in our current implementation – or by using mobile code distribution, as introduced in *Section 2.7.3*.

The forward and return paths through IP networks do not have to be identical and can change during on-going communications. We extend the same concept to the network stack. Even for connection-based protocols, the return path through the stack for the same flow is not necessarily identical to the forward path. For example, a node might initiate a connection to a remote node using TCP, but the recipient can decide to reply by means of UDP, provided that the initiator supports this feature: Whenever the initiator's address within the recipient module's context is resolvable to multiple distinct addresses, the module is free to decide which of these addresses to send its replies to.

We decided to encode the information about which modules were traversed in the sender's stack within the communication data themselves. Based on this information the receiver can iteratively demultiplex the data and forward it – akin to source routing – to the appropriate modules, i.e. derive the sender's execution flow path and traverse the matching module instances of its own stack in reverse order. The Internet stack solves the problem of demultiplexing incoming data through intrinsic (hard-coded) knowledge about where to forward the payload based on e.g. the protocol number encoded in the IP header. The stack composition system employs a similar method: Every module instance is assigned an unique identifier, from which the module class and the service type can be derived. On data reception, a demultiplexing module asks its stack composition system to

select the best-matching service in its own stack and to forward the payload to this module instance. If a module instance identifier within the stack provides an exact match for this identifier, it is chosen. Otherwise, if an instance of the same module class is found, it is selected. If neither is the case, another module offering the same service interface is selected. If the selected module is unable to parse the data or no appropriate module instance can be found, an error is returned to the demultiplexing module, which in turn may then either just drop the data or return an error message to the sender, e.g. by means of an ICMP protocol unreachable message. This approach does not prevent tunnelling or encapsulation of protocol data, as the actual encoding of the data is protocol-specific. Compatibility with existing protocols is also easy to maintain as, for example, our implementation of IPv4 encodes the needed information in the protocol field of the IP header.

Since we require that every module available to the stack composition system is instantiated at least once in every stack composition, the initiator of a communication can be sure that the message is decodable by the recipient, provided that an implementation of a compatible module is available there. This ensures that the initiator's stack composition can be modified independently of its communication partners'.

The recipient is free to decide on its own whether to use the same path through the stack when responding to incoming communications, based on its own stack configuration and module-internal decisions. Likewise, this path is not required to remain static for the duration of the communication. In our system a traffic-type based decision process can, for example, easily be placed into the stack, which channels FTP bulk data transfers through TCP, as according to the standard, but utilises UDP for the transmission of control information, even though the FTP-module invokes the same connector for both.

3.7.6 In-stack Redirectors

In-stack redirectors provide the means for the evolution engine to encode decision and branching heuristics into the stack and thus control its operations on a far more fine-grained scale then otherwise possible (see *Section 4.3*). The stack composition system provides two types of redirectors, as shown in *Figure 3.9*, threshold-based binary selectors and generic binary selectors with an associated decision function.

Threshold-based binary selectors are a generic module class which can be specialised for any arbitrary interface as required. They provide two child connectors which are bound to the same interface type the selector provides.

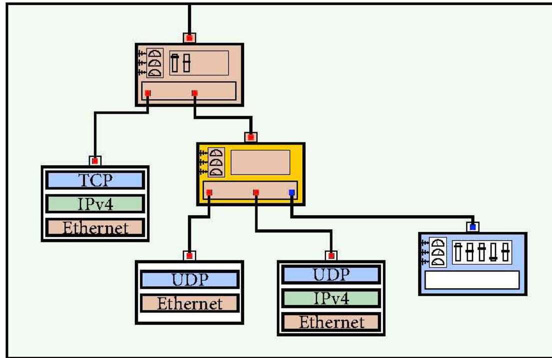


Figure 3.9: *The two types of redirectors currently available to the stack composition system are depicted here. The top-most module instance is a threshold-based redirector, which forwards all calls to one of the two connected sub-stacks depending on whether the most recent sensor measurement is above or below a threshold. The module highlighted in yellow represents a generic selector, which depends on another module (given in blue) for the actual decision process.*

All calls to the interface are directly forwarded to one of these connectors, depending on whether the data measured by the selected sensor is above or below a defined threshold. Which sensor to use, as well as the threshold, can be either statically configured at run-time or autonomously evolved by the evolution logic. A partial configuration is also possible, e.g. by specifying an interface type or by defining a range of possible sensors or thresholds for the evolution logic to experiment with. Required features for both connectors can again be independently specified.

Generic selectors externalise the decision function, but are otherwise identical to the aforementioned threshold-based selectors: The decision on which of the child connectors to use for calls to its interface is taken by decision module. For this purpose it provides another connector, which can bind to an arbitrary decision module instance that offers a predefined set of features. Every time its service interface is invoked, the selector invokes this connector, and based on the return value in the range $[0, N-1]$ forwards the call to one of its N child connectors.

Decision modules encode arbitrary decision processes that can be placed anywhere in the stack, and thus provide a simple, but powerful means for making the decision independent of the redirector's own location within the stack. Thus multiple redirectors can e.g. use the same decision mod-

ule, which inspects the packet or the flow state and decide based on these how to handle the packet. For example, a decision module might return 1 if the destination of a packet is known to support a specific protocol, and 0 if it is known not to support the protocol, and probabilistically choose either 0 or 1 otherwise, and thus non-deterministically explore the capabilities of remote nodes.

3.8 Chapter Summary

In this chapter we introduced the reasoning that guided our research into autonomous stack evolution and described the requirements that our system has to fulfil. We then detailed our system architecture, introduced the components of the system and discussed the functionality encompassed therein while focussing on their operational characteristics.

Chapter 4

Stack Evolution Logic

The stack evolution logic operates on three interdependent layers, which employ different methodologies for adaptation and differ in their temporal granularity, as illustrated by *Figure 4.1*.

Long-term adaptation is managed by the evolution engine, which generates stack blueprints and decides which modules are instantiated how often, how they they are connected to each other, and how they are configured. The machine-learning techniques described in *Section 4.1* direct the evolution of the stacks towards a higher utility, i.e. progressively adapt the stack configurations encoded in the stack blueprints towards the current situation of the environment, based on fitness estimates derived through trial-and-error experimentation. The evolution engine thus provides the core of the adaptation functionality, as it ventures to find the best possible composition of module instances and their configuration for the particular environment characteristics. It also introduces novelty into the configurations, for example by means of randomisation, mutation, or crossover.

Since we intend for the stack composition system to adapt to recurring environmental changes which happen at a faster rate than the speed the long-term evolution methodology can manage on its own, we provide another adaptation mechanism, which is able to react more quickly to such changes: The mid-term adaptation logic, described in *Section 4.2*, detects changes in the network and traffic conditions and switches to the population, i.e. pool of stack blueprints, which evolved under the most similar situation. This step further helps to ensure comparability of the trial-and-error experiments on which the evolution engine depends, as situational changes can influence the perceived fitness and invalidate the experimentally gained expertise.

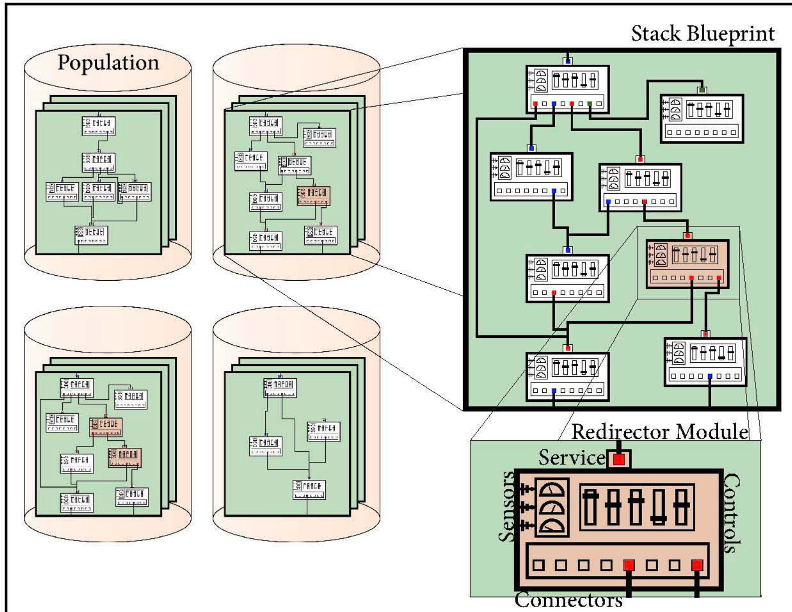


Figure 4.1: *The different scopes on which the evolution machinery operates. Short-term evolution is encompassed by redirector modules as part of the stack blueprints. These blueprints encode the stack configurations that the evolution logic creates. Blueprints are stored in populations, between which the mid-term situational classifier selects.*

The short-term adaptation logic, described in *Section 4.3*, consists of in-stack redirector and decision modules that are encoded into the stack configuration by the evolution engine. These modules encode traffic flow-based decisions and thus serve to capture and control the runtime dynamics of network operations: By being located within the stack, they take branching decisions for every invocation of the service they provide, forward all requests unmodified to one of multiple connected modules, and thus choose a different sub-stack based on e.g. the flow id, traffic class, or any other arbitrary criterion. The system thus becomes able to react instantly not only to changes in the environmental characteristics, but also to variances within a flow, without incurring the overhead needed for stack modification.

We provide several algorithms for mid- and long-term adaptation, the selection between and parametrisation of which naturally alters the char-

acteristics of the adaptation process. We discuss the implications and the possibility for dynamic on-line adaptation of their parametrisation in *Sections 4.1.4 and 4.2.3.*

4.1 Long-Term Decisions – Evolution Engine

As introduced in *Section 3.6*, the evolution engine incorporates the logic that guides the stack evolution process. Each stack blueprints encodes one stack configuration, i.e. the configuration of module instances, as well as the connections and the interaction between them. The evolution logic derives a new set of stack blueprints based on the assessed fitness of the previous generation of stack configurations. In this section we first introduce the general concepts pertaining to the evolution logic, give an overview of the general operation and characteristics of the algorithms, then thoroughly introduce the algorithms in detail, and finally describe how to select between and dynamically configure them at run-time.

4.1.1 Configuration Space & Fitness Landscape

As shown in *Figure 4.2*, every stack blueprint P can be represented by an N-dimensional vector. The control parameters and connectors of every module instance each constitute one dimension in this vector, and are encoded as a subset of $\mathbb{Z}$. The search or configuration space $\mathcal{D}_P$ on which the algorithms operate is the domain of all possible (valid and invalid) stack blueprints when expressed in the form of the aforementioned vector[2].

Whereas some of the algorithms discussed below are able to add instances to or remove them from the blueprint, we limit the minimum number of instances to one (to enforce support for sender-initiator-defined composition as discussed in *Section 3.7.5*), and the maximum to a runtime-configurable number of N_m instances for module M (to prevent degeneration of the stack). For discussion purposes we can without limitation of generality[3] assume that every blueprint contains exactly N_m instances of every module m. We thus define the configuration space as

$$\mathcal{D}_P = \mathcal{D}_{m_1}^{N_{m_1}} \times \cdots \times \mathcal{D}_{m_{|\mathcal{M}|}}^{N_{m_{|\mathcal{M}|}}} \subseteq \mathbb{Z}^{n_{m_1,\text{ctl}}} \times \mathbb{Z}^{n_{m_1,\text{ctl}}} \times \cdots \times \mathbb{Z}^{n_{m_{|\mathcal{M}|},\text{con}}},$$

[2] Some algorithms, e.g. the Evolutionary Algorithm, internally use a different but equivalent representation.

[3] As unconnected and thus superfluous module instances are removed during the stack instantiation process, we do not have to discern between the actual number of instances in the blueprint and can assume the maximum.

where $n_{m_i,\mathsf{ctl}}, n_{m_i,\mathsf{con}}$ are the number of controls and connectors defined for instance m_i, respectively.

This configuration space can quickly become huge. For example, assuming the simple case that two modules A and B are represented in a blueprint, with $n_A = 1, n_B = 2$, and where A defines one control $A_1 \in [10, 20] = \{i \mid \forall i \in \mathbb{Z} : 10 \leq i \leq 20\}$, and one connector that can bind to the service offered by B, and B defines two controls $B_1 = [0, 2], B_2 = [10, 100]$. The resulting configurations space is defined by

$$\begin{aligned}
\mathcal{D}_P &= \mathcal{D}_A \times \mathcal{D}_B^2 \\
&= \langle \text{range of } A_1 \rangle \times \langle \text{\# choices for A's connector} \rangle \times \\
&\quad \langle \text{range of } B_1 \text{ for } 1^{\text{st}} \text{ instance of B} \rangle \times \\
&\quad \langle \text{range of } B_2 \text{ for } 1^{\text{st}} \text{ instance of B} \rangle \times \\
&\quad \langle \text{range of } B_1 \text{ for } 2^{\text{nd}} \text{ instance of B} \rangle \times \\
&\quad \langle \text{range of } B_2 \text{ for } 2^{\text{nd}} \text{ instance of B} \rangle \\
&= [10, 20] \times [0, 1] \times [0, 2] \times [10, 100] \times [0, 2] \times [10, 100],
\end{aligned}$$

and encompasses $|\mathcal{D}_P| = 11 \cdot 2 \cdot 3 \cdot 91 \cdot 3 \cdot 91 = 1639638$ distinct stack configurations. Deterministic exploration would thus only be possible if the number of deployed modules and their options for configuration is severely limited, and we therefore concentrate on non-deterministic algorithms which can operate on large search spaces.

We assume that all stacks are tested under comparable conditions, i.e. that the situation, the set of conditions that define the network environment, ongoing traffic, etc., remains static. Thus we can define the optimum which we aim to attain as

$$\arg\max_P \mathsf{F}[P],$$

in other words, under this assumption the fitness depends solely on the configuration of the stack and we ideally would like to find one stack blueprint $P \in \mathcal{D}_P$ for which the fitness is maximal, or at least in close proximity of the maximum. The fitness landscape is the evaluation of $\mathsf{F}[P]$ for all possible blueprints P, i.e. the entire domain $\mathcal{D}_P$.

The performance of the evolution logics fundamentally depends on the complexity of the configuration space and the topology of the associated fitness landscape. In the following section we motivate why multiple algorithms are necessary, and in *Section 4.1.4* we discuss the performance of the individual algorithms in relation to the shape of the fitness landscape.

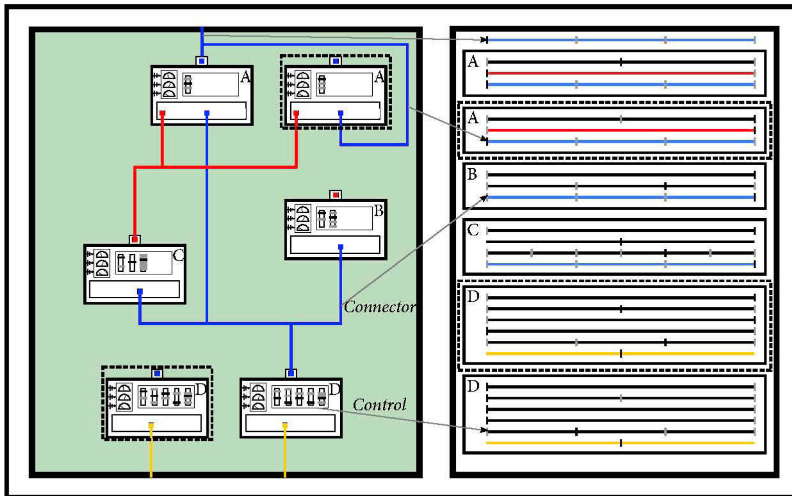


Figure 4.2: *The relation between stack blueprints and the configuration space. In this example every possible control and connector in the blueprint is represented by one of the 26 dimensions of the configuration space. Here the system was configured to include at most two instances of classes A and D, and one of classes B and D, each. Superfluous instances (marked by a dashed line) are removed when the stack is created and therefore do not consume any resources. Since we require that at least one instance of every module is present in every stack, the only instance of B is not removed from the blueprint, even though no connector refers to it. The number of possible values per dimension (grey ticks in the figure) corresponds to the number of instances that match the connector's service or the range of the control, respectively.*

No Algorithm to Rule Them All

The decision which evolution logic to use for the composition system and how to configure it has strong implications for the performance of the evolution process, as the evolution quality[は] and speed[に] of different algorithms varies widely depending on the shape of the configuration space and the fitness landscape. In fact, no one algorithm *can* be superior for all possible

[は] Here evolution quality denotes the fitness achieved by the best stack derived from any blueprint in the particular generation.

[に] Likewise, evolution speed denotes the time needed to reach a specific fitness threshold. Time is measured in generations, and the fitness of the best stack found so far is used for comparison with the threshold value.

application areas of the stack composition system.

As we already introduced in *Section 2.6.5*, Wolpert and Macready show that no algorithm can perform better than random search for all possible problems.[381] Their theorems have clear implications for our research, as the "black-box" optimisation algorithms they discuss in their paper constitute the set of algorithms from which we can choose our evolution logics. And this problem is aggravated by the fact that we necessarily – and also intentionally – limit the amount of background knowledge about the problem specification that is available to the system: Since we want the freedom to add new and remove ineffective or obsolete (micro-protocol) modules at run-time, the shape of the configuration space is unknown at design time, in fact we do not even know its dimensionality. Furthermore, we cannot make any assumptions about the other component of the algorithms domain either, as the fitness landscape depends on the run-time-defined fitness function and the knowledge of the situation in which the system is also limited and tainted by measurement noise. The very reason for using trial-and-error-based experimentation in our design is our claim that we cannot know in advance, what fitness a specific stack will achieve at run-time, even if the fitness function were known in advance.[296] **We therefore claim that no one specific evolution logic can be sufficient for all possible applications of the stack composition system.** Our framework therefore supports and includes multiple algorithms, which we introduce below.

4.1.2 Calling Conventions

All algorithms presented in this section share the same functional signature and are therefore easily interchangeable. After every complete trial of a generation of stack configurations, i.e. once the fitness of each stack corresponding to the previously generated blueprints has been experimentally determined, the **function** EVOLVE is invoked with the following parameters:

- Blueprints $\{P_1, \ldots, P_M \,|\, \forall i \in [1, M]\}$, the blueprints trialled during the previous generation,
- Fitness $\mathsf{F} : \{P_1, \ldots, P_M \,|\, \forall i \in [1, M]\} \rightarrow [0, 1]$, where $\mathsf{F}[P_i]$ denotes the fitness measured for the stack corresponding to blueprint P_i.

These functions all return the next generation of blueprints, $\{P'_i : \forall i \in [1, M']\}$, which in turn will put to experimental trial. If no initial generation is presented, it will be generated as described below for each algorithm, i.e. in most cases at random.

The encoding of the blueprints passed to and returned by these functions is identical for all algorithms. The internal representation used for deriving the next generation of blueprints, however, depends on the algorithm and is described there. The necessary steps for converting between the different representation formats is omitted, as the conversion is straightforward and a discussion would not provide any valuable insights.

4.1.3 Algorithms for Controlled Evolution

Since we are exploring fairly novel ground[ほ], we selected several textbook algorithms from the fields of machine learning and even the – at first seemingly unrelated – robot path planning, which showed promise for our purpose. We were able to apply some of these algorithms, i.e. the rule-based exploration, brute-force, and random probing, rather easily to the problem at hand, as we only had to devise an adequate representation of the configuration space. In the case of the Evolutionary Algorithm, however, we had to perform extensive modifications to derive a suitable representation and tweak the internal operation of the algorithm. One common path planning method called Rapidly-exploring Random Tree proved particularly difficult to apply, as the textbook method requires a distance metric for the configuration space, a notion of direction within it, a method for traversing a certain distance within the space, and knowledge of the goal position. We thus had to invent an appropriate metric over the stack configuration space, a way to find another configuration which is a certain distance away from a given one, yet more-or-less within an also given direction from it, and finally modify the algorithm such as to favour reaching "higher ground", i.e. configurations of higher fitness, instead of reaching a given goal position. Based on initial experimental results for these algorithms, we designed a new algorithm, which combines the advantages of both the Evolutionary Algorithm and Rapidly-Exploring Trees, and which we named the Composition Tree Search. In the remainder of this section we discuss most[219] of the implemented algorithms exhaustively, followed by a brief summary of the algorithms and their adaptation behaviour, as well as guideline of how to select between them in *Section 4.1.4*.

Evolutionary Algorithm

Evolutionary Algorithms are a class of search heuristics that are inspired by and try to mimic the process of Darwinian evolution. Evolutionary Al-

[ほ] We found surprisingly little research literature directly pertaining to algorithms for autonomous stack evolution.

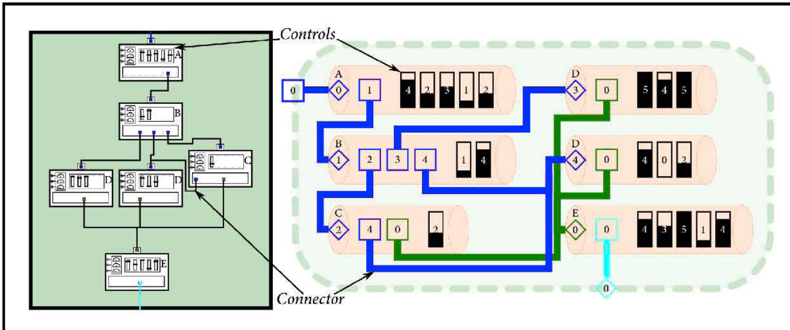


Figure 4.3: *Within the genome, chromosomes encode module instances. These chromosomes can be imagined as arrays of integer ranges each of which represents either a control (represented by a partially filled black box in the figure) or a connector (coloured boxes) that is bound to a matching service (coloured rhombus) provided by either another chromosome or the stack steering system.*

gorithms have a long history and have been applied, for example, to autonomous computing,[99] service composition,[126] or routing.[6,29,385]

Generally, evolutionary algorithms employ a population of candidate solutions, which is usually randomly initialised. Onto these a fitness function is applied which evaluates the solutions' utility, e.g. the proximity to the goal state or its performance. Based on the output of this function, the parent solutions for the next generation of the population are selected, with fitter solutions being more likely to be chosen. From these parent solutions, the next generation of the population is generated, usually by applying crossover and/or mutation.

Our decision to experiment with evolutionary algorithms for finding the best stack configuration is based on their track record as efficient methods for exploration of complex problem spaces without requiring much background knowledge: A singular fitness value suffices as signal which leads towards finding the optimum, and thus arbitrary input, e.g. user satisfaction, can be utilised. Furthermore, the approach is generic enough to be applied to many different optimisation tasks, as little knowledge of the problem structure is required, and the adaptation process basically consists of more or less random modification and recombination of arbitrary data.

Probably the most common variant of these algorithms are Genetic Algorithms, which encode the problem solution as a (often binary) string, onto which mutation and recombination are applied. Evolution Strategies in turn

use vectors of floating point numbers to encode the solution, and employ selection and mutation through addition of a normal-distributed value. The algorithm we utilise in for stack composition is based on the concept of Genetic Algorithms, but adapted to the needs of the stack composition system: The textbook Genetic Algorithms [137] we used for initial experimentation did not offer sufficient adaptation speed and produced many invalid configurations, which was detrimental to the overall adaptation speed and quality. The algorithm described in the following section is the result of our continuous evaluation, re-design, and fine-tuning, and fairly different from the original algorithm: The general concept, i.e. the use of operators like selection, mutation and crossover, conforms to common usage, but we specifically designed the representation of the stack configuration and intensively modified the operational characteristics of these operators.

Blueprint Representation The stack compositions are encoded as a genome (*Figure 4.3*) that consists of one chromosome per module instance. Genomes are encoded as a continuous vector of integers, in which the chromosomes are concatenated. Chromosomes contain the module class identifier, all control values defined for this class, as well as all of the class' connectors. Control values are limited to the range defined in the class specification. Likewise, connectors are bound by the number of chromosomes in the genome that match the requirements of this connector. In other words, a value of zero represents the first chromosome within the vector whose associated class offers the requested service identifier and the required features, as defined in the connector's specification.

Initialization The population onto which the algorithm is applied consists of a runtime-configurable number of genomes, i.e. stack composition blueprints. The genomes are initialised to contain one chromosome for every module class available. The controls and connectors are randomly initialised to values within the respective valid range.

Exploration Process Subsequent generations of genomes are derived from the previous generations in a way similar to the elitist^ roulette-wheel selection process employed in Genetic Algorithms: The genomes of the preceding generation are ordered according to their measured fitness. Then a configurable number of the genomes is directly taken over into the next generation,

^ Elitism has been shown to improve search performance, especially in the case of multi-objective Evolutionary Algorithms [306,395]

starting with the fittest genome. For all remaining genomes of the new generation, two parent genomes from the preceding generation are randomly chosen. The likelihood of being selected is proportional to the measured utility, meaning that genomes which achieved a higher fitness value are more likely to be selected as parents. This process is described by the pseudo code presented in *Algorithm 4.1*, where $M_e \leq M$ denotes the size of the elite.

Algorithm 4.1: Evolutionary Algorithm, Exploration Process

```
function EVOLVE_EA(Blueprints {P_1, ..., P_M}, Fitness F)
    sort({P_1, ..., P_M}, F)                         ▷ ensure F[P_i] ≥ F[P_{i+1}]
    for i ← 1 → M_e do                               ▷ copy elite configurations
        P'_i ← P_i
    end for
    i ← M_e + 1
    while i ≤ M do
        i_0 ← rws({P_1, ..., P_M})                   ▷ select parents using
        i_1 ← rws({P_j | 1 ≤ i ≤ M, j ≠ i_0}         ▷ roulette-wheel selection
        P'_i, P'_{i+1} ← meiosis(P_{i_0}, P_{i_1})   ▷ produce two children
        P'_i ← mutate(P'_i)                          ▷ perform mutation
        P'_{i+1} ← mutate(P'_{i+1})
        i ← i + 2
    end while
    return {P'_1, ..., P'_M}
end function
```

The algorithm utilises the roulette-wheel selection algorithm shown in **function** RWS in *Algorithm 4.2*, for determining the parent genomes from which two child configurations will be produced.

We originally modelled the process of deriving a child genome from the two parents after the sexual reproduction (meiosis) employed by eukaryotes, but the resulting algorithm performed poorly during our initial experiments. We managed to improve the performance by adapting the algorithm as shown in **function** MEIOSIS in *Algorithm 4.3*.

At first we distribute the chromosomes of the parents between the two children. This step is performed by **function** DISTRIBUTE_CHROMOSOMES, *Algorithm 4.4* and maintains the proportions of each service type within the parent: If the first parent contains $N_{0,c}$ module instances that provide service c, and the second parent contains $N_{1,c}$, then the first child will get $N_{0,c}$ of these modules, which are randomly chosen from both parents, and the

second child will receive the remainder.

Algorithm 4.2: Roulette-Wheel Selection

function RWS(Blueprints $\{P_1, \ldots, P_M\}$, Fitness F)
  Preconditions: $\mathsf{F}[P_i] \geq \mathsf{F}[P_{i+1}], \forall i \in \mathbb{Z}, 1 \leq i < M$
  $f_s \leftarrow \sum_i \mathsf{F}[P_i]$
  $r \leftarrow \text{random}([0, f_s))$
  $f_c \leftarrow 0$
  $i \leftarrow 1$
  while $f_c + \mathsf{F}[P_i] < r$ **do**
    $f_c \leftarrow f_c + \mathsf{F}[P_i], \quad i \leftarrow i + 1$
  end while
  return i
end function

The effects of this step are comparable to a high-probability crossover in Genetic Algorithms: The intention is to exchange and re-combine module instances among two well-performing parents: If different settings for a module offer high performance, their recombination might be beneficial. Likewise, if the settings are sufficiently similar, the effect will be negligible.

Algorithm 4.3: Evolutionary Algorithm, Meiosis

function MEIOSIS(Blueprints P_0, P_1)
  $P_0', P_1' \leftarrow$ distribute_chromosomes(P_0, P_1)
  for all $m \in$ moduleclasses **do**
    populate(P_0', m) ▷ *randomly add or*
    populate(P_1', m) ▷ *remove chromosomes*
  end for
  crossover(P_0', P_1') ▷ *perform crossover on chromosomes*
  fix_connectors(P_0') ▷ *make sure connectors point to*
  fix_connectors(P_1') ▷ *chromosomes within the same genome*
  return P_0', P_1'
end function

Without the constraint to maintain the proportions of the instance distribution, the adaptation towards solutions that employ multiple in-stack redirectors (see *Section 4.3.1*) would need far longer, as the expected value

of redirectors per child would be half the sum of the redirector count of the parents.

Algorithm 4.4: Evolutionary Algorithm, distribution of chromosomes amongst the offspring, ensuring that the number of chromosomes per module class stays the same as for the parents

```
function DISTRIBUTE_CHROMOSOMES(Blueprints P_0, P_1)
    P'_0 ← ∅
    P'_1 ← ∅
    for all m ∈ {class(γ)|γ ∈ P_0 ∪ P_1} do          ▷ class(γ) denotes the
        Γ_0 ← {γ|γ ∈ P_0 ∧ class(γ) = m}              ▷ module class represented
        Γ_1 ← {γ|γ ∈ P_1 ∧ class(γ) = m}              ▷ by chromosome γ
        Γ ← Γ_0 ∪ Γ_1
        Γ'_0 ← ∅
        for i ← 1 → |Γ_0| do
            γ ← random(Γ)
            Γ ← Γ \ {γ}
            Γ'_0 ← Γ'_0 ∪ {γ}
        end for
        P'_0 ← P'_0 ∪ Γ'_0
        P'_1 ← P'_1 ∪ Γ
    end for
    return P'_0, P'_1
end function
```

In the next step two-point crossover (see **function** CROSSOVER, *Algorithm 4.5*) between randomly selected chromosomes that represent the same module class is performed. The start and end position of the crossover step within the chromosome is randomly chosen, then the contents of the control and connector values encoded in between are exchanged. Thus partial recombination of chromosome settings is encouraged.

After crossover is performed connectors may point to chromosomes that are now part of the other child's genome. We therefore need to modify the affected connectors so that they point to a valid chromosomes within the same genome, i.e. those which provide the same service type. For this purpose we calculate the distance between the connector's original destination chromosome within the parent genome and all chromosomes of the same type in the child, then redirect the connector to a chromosome in the child for which the distance is minimal. We define the distance as the sum of the

deltas between the control and connector values, which is iteratively calculated by **function** FIX_CONNECTORS, *Algorithm 4.6.*

Algorithm 4.5: Evolutionary Algorithm, crossover

function CROSSOVER(Blueprints P_0, P_1)
  $P \leftarrow P_0 \cup P_1$
  $U \leftarrow \emptyset$
  for all $\gamma \in P \setminus U$ **do**
    $\Gamma \leftarrow \{\gamma' | \gamma' \in P, \gamma' \neq \gamma, \text{class}(\gamma') = \text{class}(\gamma)\}$
    $\gamma' \leftarrow \text{random}(\Gamma)$
    $P \leftarrow P \setminus \{\gamma'\}$
    if $\text{random}([0, 1)) \leq p$ **then**
      $r_0 \leftarrow \text{random}([1, |\text{controls}(\gamma)|])$
      $r_1 \leftarrow \text{random}([r_0 + 1, |\text{controls}(\gamma)|])$
      $\text{swap}(\gamma, \gamma', r_0, r_1)$
    end if
  end for
end function

Algorithm 4.6: Evolutionary Algorithm, connector fix-up

function FIX_CONNECTORS(Blueprint P)
  for all $\gamma \in P$ **do**
    for all $c \in \text{connectors}(\gamma)$ **do**
      $\gamma' \leftarrow \text{target}(c)$
      if $\gamma' \notin P$ **then**
        $\text{target}(c) \leftarrow \arg\min_{\gamma_i \in P} \Delta(\gamma, \gamma_i)$
      end if
    end for
  end for
end function

Afterwards additional chromosomes are randomly added or removed, depending on the number of unreferenced chromosomes of the same module class already present within the genome. Unreferenced chromosomes are those which are not targeted by any connector. This step was again taken for

the benefit of in-stack redirectors, the corresponding pseudo code is given in **function** POPULATE, *Algorithm 4.7*: If the probability of inclusion of a redirector in the blueprint were to be fixed to $\mathcal{P}_r$, the probability of a blueprint encoding N-ary decision trees of depth M in the worst case obviously becomes a marginal $\mathcal{P}_r^{N^M}$.

Algorithm 4.7: Evolutionary Algorithm, random addition or removal of chromosomes

```
function POPULATE(Blueprint P, Chromosome m)
    U ← {γ|γ ∈ P ∧ class(γ) = m∧
            ∄γ′, ∃c ∈ connectors(γ′), target(c) = γ}
    r ← random([0, 1))
    if |U| = 0 ∧ r ≤ 0.8 then
        add_chromosome(P, m)
    else if |U| = 1 ∧ r ≤ 0.1 then
        add_chromosome(P, m)
    else if |U| = 2 then
        if r ≤ 0.05 then
            P ← P ∪ {random_chromosome(m)}
        else if r ≤ 0.2 then
            P ← P \ {random(U)}
        end if
    else if |U| ≥ 3 then
        if r ≤ 0.05/|U|² then
            P ← P ∪ {random_chromosome(m)}
        else if r ≤ 0.2|U| then
            P ← P \ {random(U)}
        end if
    end if
end function
```

Finally the control and connector values are mutated with a configurable probability as follows (see **function** MUTATE, *Algorithm 4.8*). Nominal controls are set to a new random value, the only constraints for which are to remain within the control's range and to differ from the previous value. Continuous controls are mutated according to a Gaussian-distributed random variable $X \sim \mathcal{N}(\mu, \sigma^2)$, with μ set to the previous control value, and the runtime-configurable σ scaled in relation to the length of the range of valid values for the control, to ensure that values in the vicinity of the previous

value are more likely to be selected. We further ensure that the result lies within definition range of the control and does not match the previous value.

Algorithm 4.8: Evolutionary Algorithm, mutation

```
function MUTATE(Blueprint P, Chromosome m)
    for all γ ∈ P do
        for all c ∈ controls(γ) do
            r ← random([0, 1))
            if r ≤ P_M then
                if type(c) = continuous_control then
                    c ← rndgauss(c, σ^2)          ▷ normal-distributed RNG
                else
                    c ← random(D_c)               ▷ uniformly distributed RNG
                end if
            end if
        end for
    end for
    for all γ ∈ P do
        for all c ∈ connectors(γ) do
            r ← random([0, 1))
            if r ≤ P_M then
                c ← random(D_c)
            end if
        end for
    end for
end function
```

The Evolutionary Algorithm thus tries to model the operation of evolution and selection of the fitness, which in nature proved its effectiveness for undirected adaptation if the time-frame is long enough. During our experiments, the general potential of this method became apparent, even though other methods surpassed it in terms of quality and speed of the adaptation process. While this algorithm tends to perform poorly when applied to binary decision tasks, and if incorrectly configured can optimise towards local optima, our research shows that it can fairly quickly find a *better* solution than the current one, and operate sufficiently well without having to be adapted to a specific problem space, and therefore fits well into our design requirements, as detailed in *Section 3.2*. The adaptation behaviour of this algorithm during our experiments was often inferior even to that of the ran-

dom probing algorithm, e.g. for the scenario described in *Section 6.1.3*, but these results do not imply a general inferiority.

Rapidly-Exploring Random Tree

Rapidly-Exploring Random Trees[209] provide an approach for high-dimensional path planning tasks that is especially efficient when the search space involves (differential) state constraints or inaccessible regions. We decided to explore the use of path planning algorithms for the task at hand, because we assumed that in many cases gradual adaptation of a known good initial stack configuration can quickly lead to a better stack configuration than the current. The Rapidly-Exploring Random Tree in particular is known to quickly find a path towards a goal configuration even if the search space is huge. The algorithm operates by incrementally updating a tree such as to quickly reduce the distance to a randomly selected point in the search space. It thus implements a Monte-Carlo biasing search into the largest Voronoi regions, as shown in *Figure 4.4*, i.e. the largest unexplored areas in the search space are the most likely to be visited.

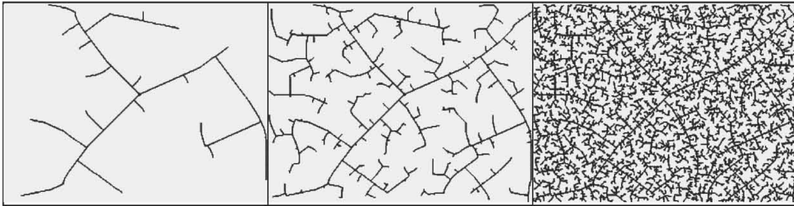


Figure 4.4: *Rapidly-Exploring Random Trees rapidly fill the available search space, as the likelihood of exploring the largest Voronoi regions in the graph is high.*

The original method can not directly be applied to the problem at hand, as it requires a non-convex search space to traverse. Deriving a distance definition based on the components of our stack configurations, i.e. instances, controls and connectors, is obviously non-trivial. We had to design a configuration space representation, the corresponding metrics and thus devised a means for calculation of the distance between stack configurations which attributes a weight to changes that is proportional to their impact on operations: For example, replacing one module instance with one of another class is weighed higher (and thus more distant) than a slight modification of a control value within the same instance.

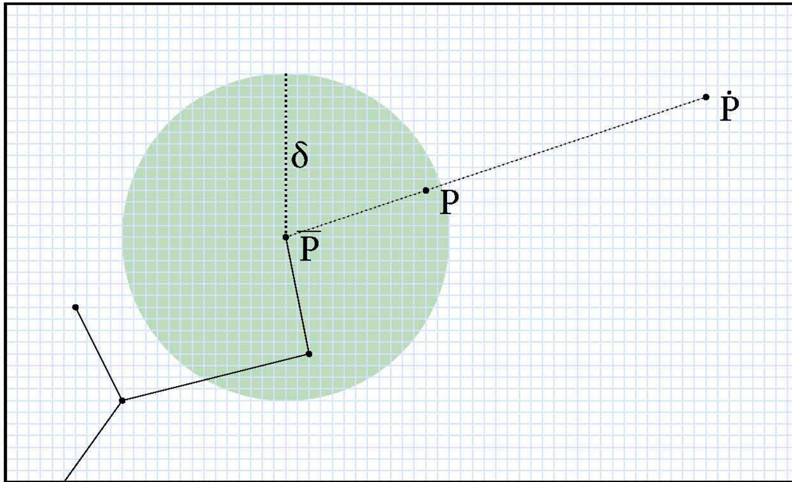


Figure 4.5: *The process of finding a new position P to explore and add to the Rapidly-Exploring Random Tree. $\dot{P}$ is chosen at random from the not yet visited positions in the search space. $\bar{P}$ is the closest point to $\dot{P}$, that is already part of the tree. The new position to explore P is then found by moving at most δ from $\bar{P}$ into the direction of $\dot{P}$. The grid denotes valid positions in the search space. As δ is variable, the members of the tree are not equidistant.*

Blueprint Representation Every blueprint is interpreted as a vector into an N-dimensional configuration space as introduced in *Section 4.1.1*. The Rapidly-Exploring Random Tree implementation is working on a fixed dimensionality defined by the maximum number of instances per module, i.e. module instances cannot be added or removed. The configurations are therefore likely to contain an encoding for superfluous instances, i.e. those instances that are not accessible through a connector and are also not the sole instance of a class. Such instances are silently omitted when the corresponding stack is instantiated.

Initialization The Rapidly-Exploring Random Tree starts with a population consisting of only one stack configuration that is randomly initialised and which acts as the root of the tree.

Exploration Process Starting from the root, the tree is built by iterating over all previously visited configurations (in the order in which they were

added to the tree) and adding a new leaf node which is connected to the nearest current member of the tree. During every iteration of the search process in the original Rapidly-Exploring Random Tree algorithm, a new position $\dot{P}$ within the configuration space is chosen at random. The node $\bar{P}$ of the tree that lies nearest to $\dot{P}$ is selected. A new leaf is then added at position P, which lies at a distance of less or equal to a pre-configured δ when moving from $\bar{P}$ towards $\dot{P}$, i.e.

$$P := \bar{P} + \min\left(\delta, \left|\dot{P} - \bar{P}\right|\right) \cdot \frac{\dot{P} - \bar{P}}{|\dot{P} - \bar{P}|}$$

when moving in Euclidean space. The correlation between $\bar{P}$, $\dot{P}$, and P is shown in *Figure 4.5*

We adapted this algorithm for the stack composition problem as follows: A new randomly initialised vector $\dot{P} = \langle \dot{p}_1, \ldots, \dot{p}_N \rangle$ is generated, and P is assigned the nearest vector $\bar{P}$ within the M best-performing blueprints of the previous generation(s). P is then modified such as to lie δ closer to $\dot{P}$. Thus the search process is described by the pseudo code for **function** EVOLVE_RRT in *Algorithm 4.9.*

Algorithm 4.9: Rapidly-Exploring Tree, Exploration Process

```
function EVOLVE_RRT(Blueprints {P_1, ..., P_M}, Fitness F)
    for i ← 1 → M do
        Ṗ ← randomize()                ▷ select random pos. in configuration space
        P̄ ← arg min_{P_i} Δ(P_i, Ṗ)    ▷ find vertex in tree that lies nearest to Ṗ
        P'_i ← advance(P̄, Ṗ, δ)        ▷ modify P̄ such as to lie δ closer to Ṗ
    end for
    return {P'_1, ..., P'_M}
end function
```

For distance calculation we use the metric Δ over all weighted control and connector values:

$$\Delta(P, P') = \sum \Delta_k(p_k, p'_k)$$

$$\Delta_k(p_k, p'_k) = \begin{cases} \dfrac{|p_k - p'_k|}{|\mathcal{D}_k|}\omega_k & \text{if } k \text{ is a continuous control} \\ \omega_k & \text{otherwise} \end{cases}$$

where ω_k denotes the weight and $\mathcal{D}_k$ the domain of definition for control or connector k. The advancement from $\bar{P}$ towards $\dot{P}$ is performed according to the pseudo code for **function** ADVANCE given in *Algorithm 4.10*.

Algorithm 4.10: Rapidly-Exploring Tree, local random search

```
function ADVANCE(Blueprints P̄, Ṗ = ⟨ṗ_1, ..., ṗ_N⟩ , Distance δ)
    P = ⟨p_1, ..., p_N⟩ ← P̄
    K ← {1, ..., N}
    while δ > 0 ∧ K ≠ ∅ do            ▷ advance at least δ from P̄ towards Ṗ
        k ← random(K)
        K ← K \ {k}
        if ṗ_k ≠ p_k then
            if type(k) = continuous_control then
                d ← sig(ṗ_k − p_k)
                δ' ← random ([0, min (|ṗ_k − p_k| , δ)])
                p_k ← p_k + dδ'
                δ ← δ − δ_k ω_k / |D_k|
            else
                p_k ← random (D_k \ {p_k})
                δ ← δ − ω_k
            end if
        end if
    end while
    return P
end function
```

In our initial tests this non-deterministic search method proved to perform rather well, being on par with the Evolutionary Algorithm described in the previous section. While preparing the experiments we describe in this document, however, the this method miserably failed to lead to improvements in the stack performance. Since the Composition Tree Search incorporates some of the ideas of the Rapidly-Exploring Random Tree, we decided to nevertheless include the algorithm's description in here, but did not pursue the experimental evaluation any further for the time being.

Composition Tree Search

Inspired by the aforementioned Rapidly-Exploring Random Trees, we devised a search algorithm which additionally integrates some of the character-

istics of the Evolutionary Algorithm. The original Rapidly-Exploring Random Tree algorithm assumes that the likelihood of the goal position is the same anywhere in the search space. Since we found the fitness landscape of most scenarios we explored to not be too chaotic, i.e. since fitness changes between neighbouring configurations are gradual, we biased our approach to towards exploring the vicinity of fitter configurations. We then gradually adapted this concept based on the experience obtained through trial-and-error experimentation. The version presented below is the final result of our adaptation.

Algorithm 4.11: Composition Tree Search, Exploration Process

function EVOLVE_CTREE(Blueprints $\{P_1, \ldots, P_M\}$, Fitness F)
    Preconditions: $\mathsf{F}[P_i] \geq \mathsf{F}[P_{i+1}], \forall i \in \mathbb{Z}, 1 \leq i < M$
    for $i \leftarrow 1 \rightarrow \lfloor M_e \rfloor$ **do** ▷ *copy elite configurations*
        $P'_i \leftarrow P_i$
    end for
    if $i < M_e$ **then** ▷ *handle fractional* M_e
        $r_{max} \leftarrow M_e - i_r$
        $r \leftarrow$ random $([0, r_{max}))$
        if $r < r_{max}$ **then**
            $P'_i \leftarrow P_i, \quad i \leftarrow i + 1$
        end if
    end if
    while $i \leq \lfloor M_e + M_s \rfloor$ **do** ▷ *search in the vicinity of existing nodes*
        $P'_i \leftarrow$ local_search$(\{P_1, \ldots, P_M\}), \quad i \leftarrow i + 1$
    end while
    if $i < M_e + M_s$ **then** ▷ *handle fractional* M_s
        $r_{max} \leftarrow M_e + M_s - i$
        $r \leftarrow$ random $([0, r_{max}))$
        if $r < r_{max}$ **then**
            $P'_i \leftarrow$ local_search$(\{P_1, \ldots, P_M\}), \quad i \leftarrow i + 1$
        end if
    end if
    while $i \leq M$ **do** ▷ *initialise random configurations*
        $P'_i \leftarrow$ randomize$(), \quad i \leftarrow i + 1$
    end while
    for $i \leftarrow 1 \rightarrow M$ **do** ▷ *adjust fitness values to*
        $\mathsf{F}[P_i] \leftarrow 0.5(1 - w_f) + w_f \cdot \mathsf{F}[P_i]$ ▷ *reflect reduced confidence*
    end for
    return $\{P'_1, \ldots, P'_M\}$
end function

Blueprint Representation Identically to the Rapidly-Exploring Random Tree, every blueprint is again interpreted as a vector into the N-dimensional configuration space as defined in *Section 4.1.1*, containing the maximum number of instances allowed per module.

Initialization M starting positions P_i within the search space are randomly chosen, to reduce the likelihood of ending in up in a particularly deficient starting position and thus improve the speed of adaptation. The tree is initialised as the minimum spanning tree $\mathbf{T} = \text{mst}(P_1, \ldots, P_M)$.

Exploration Process The exploration process creates the configurations using a mixture of three independent techniques. Firstly, similarly to the Evolutionary Algorithm described above, a certain number of configurations, the elite, can be taken over from the previous generation. Secondly, configurations can be generated at random. And, lastly, they can be generated through limited modification of an existing configuration, in a process which can be likened to a randomised local search. The number of configurations generated by each of these methods is defined through independently configurable parameters. The encoding of the parameters is such that an amount of non-determinacy can be introduced, if so intended. M is the number of configurations to produce in total. M_e denotes the elite size in its integral part, as well as the probability of increasing the elite size by one, which is encoded in the fractional part. M_s defines the number of search operations for new positions to perform, in combination with the probability of producing one more configuration in this way. For example, setting $M=4, M_e=1, M_s=2.7$ defines an elite size of exactly 1, and with probability 0.7 will produce two configurations using the search process and one randomly. With probability 0.3, however, it will produce three configurations through searching and none randomly. The pseudo code for **function** EVOLVE_CTREE in *Algorithm 4.11* details this process.

Searching for new positions is performed as follows (also see *Figure 4.6*). One of the existing nodes of the tree, $\bar{P}$, is chosen through roulette-wheel selection based on the fitness determined for the associated stack compositions (see pseudo code in *Algorithm 4.2*).

The new position $P = \langle p_1, \ldots, p_N \rangle$ is then iteratively generated through modification of $\bar{P}$. The distance δ to advance from $\bar{P}$, is determined by the associated fitness value. A lower fitness results in a higher δ according to the formula $\delta \sim \mathcal{N}\left(1 - \mathsf{F}[\bar{P}], {}^1\!/\!_4(1 - \mathsf{F}[\bar{P}])\right)$, where $\mathcal{N}$ is the normal distribution, and $\mathsf{F}[P]$ is the fitness measured at $\bar{P}$.

The modification is performed by one of the two methods described below, between which the algorithm randomly chooses according to the preconfigured probability $\mathcal{P}_{mod}$. The first method is identical to the one described for the Rapidly-Exploring Random Tree, i.e. it selects a random configuration $\dot{P}$ and move towards it from $\bar{P}$, as defined in *Algorithm 4.10*. The second method operates by repeatedly selecting a not yet adjusted connector

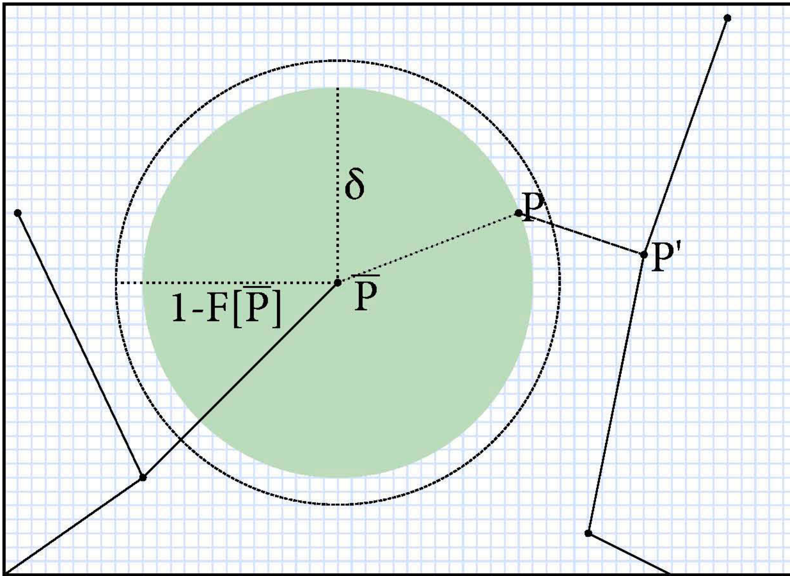


Figure 4.6: *Local random search process performed by the Composition Tree Search algorithm. The position $\bar{P}$ is chosen randomly, with probability relative to its fitness* $\mathsf{F}(\bar{P})$. *A new point P is created at distance δ from $\bar{P}$, and connected to the closest point already in the tree, which in this example is P'.*

or control value, i.e. dimension k. Continuous controls are set to a random value p_k for which $0 < |p_k - \bar{p}_k| \leq \delta$ holds. For nominal controls and connectors, the new value is randomly selected from $\mathcal{D}_k \setminus \{p_k\}$. Thus the exploration of the every configuration within a radius of δ around $\bar{P}$ is equally likely. The weighted amount of change is then subtracted from delta. This process is then repeated until $\delta \leq 0$ or all dimensions have been adjusted. The pseudo code for **function** LOCAL_SEARCH in *Algorithm 4.12*, where $\mathcal{D}_k$ denotes the domain of definition for dimension k, explains this process.

The Composition Tree Search is to best of our knowledge the first algorithm specifically developed for the problem of autonomous stack evolution by means of on-line experimentation. During our experiments, some of which are described in *Section 6*, it exhibited the best precision of adaptation of all the algorithms we explored, but further improvements by choosing a parametrisation of the algorithm that is optimised to the problem space are probably possible. While dynamic on-line parameter adaptation is possible and appears promising, our results so far are inconclusive and we do not

know when and how to perform this process most effectively. Further research in this area therefore seems advisable.

Algorithm 4.12: Composition Tree Search, local random search

function LOCAL_SEARCH(Blueprints $\{P_1, \ldots, P_M\}$) ▷ *search in the*
  $i \leftarrow \text{rws}(\{P_1, \ldots, P_M\})$ ▷ *vicinity of existing nodes*
  $\bar{P} \leftarrow P_i$
  $\delta \leftarrow \text{rndgauss}\left(1 - \mathsf{F}[\bar{P}], \frac{1}{4}(1 - \mathsf{F}[\bar{P}])\right)$ ▷ *normal-distributed RNG*
  if $\text{random}([0,1)) \leq \mathcal{P}_{mod}$ **then**
    $\dot{P} \leftarrow \text{randomize}()$ ▷ *move towards a random position*
    $P \leftarrow \text{advance}(\bar{P}, \dot{P}, \delta)$
  else
    $P \leftarrow \bar{P}$
    $K \leftarrow \{1, \ldots, N\}$
    while $\delta > 0 \wedge K \neq \emptyset$ **do** ▷ *advance at least δ from $\bar{P}$*
      $k \leftarrow \text{random}(K)$ ▷ *uniformly distributed RNG*
      $K \leftarrow K \setminus \{k\}$
      if $\text{type}(k) = \mathsf{continuous_control}$ **then**
        $p_k \leftarrow \text{random}\left(\left[\max\left(\min(\mathcal{D}_k), \bar{p}_k - \delta\frac{|\mathcal{D}_k|}{\omega_k}\right), \min\left(\max(\mathcal{D}_k), \bar{p}_k + \delta\frac{|\mathcal{D}_k|}{\omega_k}\right)\right]\right)$
        $\delta_k \leftarrow |p_k - \bar{p}_k|$
        $\delta \leftarrow \delta - \delta_k\frac{\omega_k}{|\mathcal{D}_k|}$
      else
        $p_k \leftarrow \text{random}\left(\mathcal{D}_k \setminus \{\bar{p}_k\}\right)$
        $\delta \leftarrow \delta - \omega_k$
      end if
    end while
  end if
  return P
end function

Stochastic Hill-Climbing

To enable the users of the system to leverage the benefits of knowledge they have about the application scenario, the composition system provides rule-based exploration methods, which are only adequate for specific situations, but tend to provide superior evolution speed there. One such algorithm is

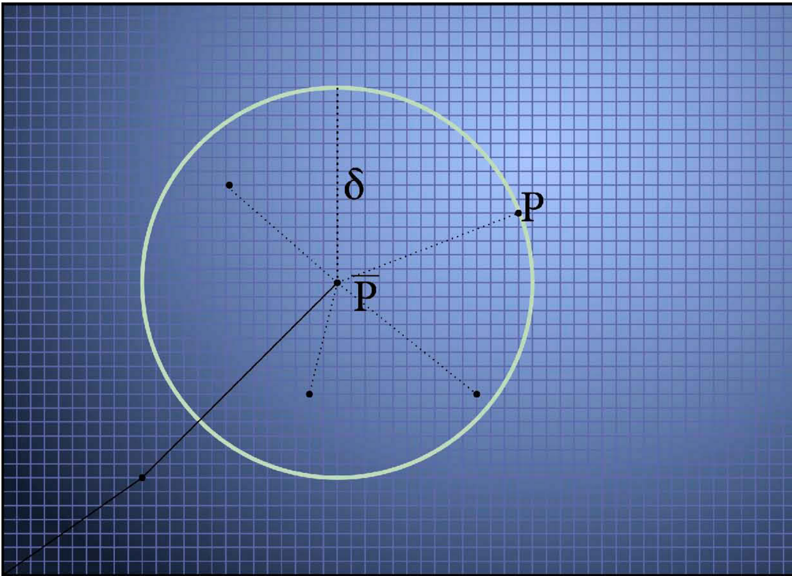


Figure 4.7: *Stochastic hill-climbing probes probes the vicinity of the currently known best configuration $\bar{P}$, until it finds a better one, in this example denoted by P.*

the common stochastic hill-climbing, which we describe in here. Since it is only usable for a limited set of scenarios where the fitness landscape is free of local maxima or ridges, we do not provide experiments for this particular algorithm. Since this method does however quickly arrive at a good solution whenever the aforementioned characteristics are met, we still considered discussing it worthwhile.

Blueprint Representation The blueprint is represented by the high-dimensional integer vector defined in *Section 4.1.1*. This definition naturally implies that only a fixed number of instances for every class can be contained in the blueprint, which is an intentional limit to the possible number of exploration steps: The hill-climbing algorithm is intended as a means to quickly find the optimal solution in a restricted set of problem classes, for more complex problems one of the other algorithms would probably be better suited.

Algorithm 4.13: Exploration Process of Stochastic Hill-Climbing, one of the Rule-based Exploration algorithms

```
function EVOLVE_HILL_CLIMB(Blueprint {P}, Fitness F)
    Persistent:   P_best, the best blueprint found so far
    Persistent:   P_known, the set of all previously trialled blueprints
    if F[P] > F[P_best] then
        P_best ← P
    end if
    while true do
        r ← random([0, 1))
        if r > 𝒫_noise then
            P' ← P_best                                  ▷ P' = ⟨p'_1, ..., p'_N⟩
        else                                 ▷ choose another configuration instead
            P' ← random(P_known)                  ▷ of the best with low probability
        end if
        d ← random({0, 1}^N \ {0}^N)                     ▷ d = ⟨d_1, ..., d_N⟩
        for k ← 1 → N do
            if d_k ≠ 0 then
                if type(k) = continuous_control then
                    p'_k ← random([max(min(𝒟_k), p'_k − δ),
                                   min(max(𝒟_k), p'_k + δ)])
                else
                    p'_k ← random(𝒟_k \ {p'_k})
                end if
            end if
        end for
        if P' ∉ P_known then
            P_known ← P_known ∪ {P'}
            return {P'}
        end if
    end while
end function
```

Initialization A runtime-configurable number of blueprint vectors is randomly initialised from within the aforementioned blueprint domain. Each of these vectors represents one point within the search space. Standard hill-climbing operates on only one such vector, however, depending on the optimisation problem, starting to search from several random locations can lead

to better results.

Exploration Process The search itself is performed by randomly choosing one of these vectors, with probability equal to the fitness the stack that corresponds to this vector achieved during the trial phase. One dimension of the vector is then randomly modified according to a Gaussian distribution centred at the current value for this dimension. The pseudo code for the stochastic hill-climbing method is presented in **function** EVOLVE_HILL_-CLIMB in *Algorithm 4.13*.The adaptation speed of this algorithm can sometimes be improved by adapting the step width[263] according to the currently achieved fitness, i.e. to move further away from the current position if the fitness is low than otherwise, or simple over time. Another approach is simulated annealing, which selects an inferior candidate position instead of superior one according to a transition probability that is based on the fitness delta and the inverse of time.[194] We discuss this possible for some of the other algorithms in *Section 4.1.4*.

Brute-force

For testing and debugging purposes, as well as for mapping the fitness landscape and determining finding the optimal stack configuration, we implemented a brute-force evolution logic, which deterministically iterates through all possible stack configurations, and which can obviously only be used when the configuration space is small.

Blueprint Representation The blueprint representation is again identical to the one introduced in *Section 4.1.1*.

Initialization The initial configuration represents the minimal vector, i.e. all controls are set to the lowest possible value and all connectors point to the first matching module instance.

Exploration Process The search process consists of iterating through all possible values for all dimensions in turn and can most easily be described by the pseudo code for **function** EVOLVE_BF, given in *Algorithm 4.14*.

Algorithm 4.14: Brute-Force Search, Exploration Process

```
function EVOLVE_BF(Blueprint {P})
    done ← false
    for k ← 1 → N do
        if done then
            p'_k ← p_k
        else
            if p_k + 1 = max D_k then
                p'_i ← min D_k
                if k = N then
                    Terminate                ▷ final configuration generated
                end if
            else
                p'_k ← p_k + 1
                done ← true
            end if
        end if
    end for
    return {P' = ⟨p'_1, ..., p'_N⟩}
end function
```

Random Probing

This method constitutes another rather simple approach to searching for a well-performing stack configuration. It can be considered a non-deterministic variant of the previously described brute-force search, as it randomly explores a yet untested configuration.

Blueprint Representation The configuration space is again defined by an N-dimensional vector introduced in *Section 4.1.1*.

Initialization Random probing starts with one randomly generated configuration.

Exploration Process The search process consists solely of randomly generating a configuration that has not yet been tried and re-use of the best configuration found so far, as shown in **function** EVOLVE_RND in *Algorithm 4.15*.

Algorithm 4.15: Random Probing, Exploration Process

function EVOLVE_RND(Blueprint $\{P\}$, Fitness F)
  Persistent: P_{best}, the best blueprint found so far
  Persistent: P_{known}, the set of all previously trialled blueprints
  if $\mathsf{F}[P] > \mathsf{F}[P_{\text{best}}]$ **then**
    $P_{\text{best}} \leftarrow P$
  end if
  if $|\mathcal{D}_P| = |P_{\text{known}}|$ **then** ▷ *all configurations tried*
    return $\{P_{\text{best}}\}$
  end if
  $P' \leftarrow \text{randomize}(\mathcal{D}_P \setminus P_{\text{known}})$
  $P_{\text{known}} \leftarrow P_{\text{known}} \cup \{P'\}$
  return $\{P'\}$
end function

We originally intended this algorithm to serve as the baseline for our experiments, to which the results of the other, less simplistic algorithms are compared. In our experiments, the random probing algorithm did however prove to be surprisingly effective, in some cases even exhibiting better performance then the Evolutionary Algorithm, as described in detail in *Section 6*.

4.1.4 Selecting the Evolution Logic

As discussed in *Section 4.1.1*, no one algorithm can deliver performance superior to random search for all possible application areas of the stack composition system. In this section we nevertheless try to formulate a few tentative guidelines on how the shape of the configuration space and the fitness landscape influences the behaviour of the algorithms, and how to select the evolution logic to use if at least some characteristics of the optimisation problem are known at design time. Afterwards, we will elaborate on the possibility of autonomous selection and (re-)parametrisation of this algorithm at run-time.

Summary of Experiences and Guideline for Off-line Selection

Due to the huge amount of possible applications, possible fitness functions, protocols and other modules available for configuration and composition,

compared to the necessarily limited number of experiments – some of which are described in *Chapter 6* – we were able to perform[ヒ], our insights into the algorithms' performance cannot be generalised. Nevertheless, we decided to present a few informal and subjective guidelines for choosing the evolution logic for a specific problem set, based on our experiments, as we found literature research to not be very fruitful: Most discussions present mutually incompatible optimisations for specific application areas,[140,177] or are too general for our purpose.[264] And as e.g. Eiben points out, the quest to find generally-applicable, yet (close-to-)optimal parameter settings a-priori is a lost cause altogether.[102]

During the experiments, our own Composition Tree Search algorithm in general exhibited the best performance. If little background knowledge of the problem space is available, we therefore recommend this algorithm. Evolutionary Algorithms can be applied to complex and not well-understood optimisation tasks,[26,110,137] but performed rather poorly during our experiments, most likely due to the low population size we were forced to use, as discussed in *Section 6.3*. If the fitness landscape is smooth, random probing can be rather effective. And if the configuration space is small but rough, an exhaustive search is advised. The Rapidly-Exploring Random Tree performed so poorly in our experiments, i.e. got frequently caught in local maxima, that we had to exclude the results from our discussion. We still include the description of this algorithm in here, as we leveraged some of its design aspects for our own Composition Tree Search.

As expected and apparent in our experiments, the parametrisation of the evolution logic has a huge impact on its performance. While we tried to choose the most effective parametrisation for the algorithms in our experiments, our decision here was also based on limited trial-and-error experimentation, thus the optimality of our settings is far from guaranteed.

We were only able to experimentally confirm that the common general considerations, e.g. those made by Bäck[24,25,26] for Evolutionary Algorithm, also apply to the algorithms we tested: Apart from the truly random probing algorithm, all other methods provide one or multiple settings which affect the maximum step width and the probability of exploring a randomly chosen state. In particular settings related the step width are very dependent on the shape and size of the configuration space, as high step widths enable the algorithms to traverse large spaces quickly, but impact their precision, i.e. the best solution is less likely to be found. The optimal setting for the probability of exploring a randomly chosen configuration in turn depends

[ヒ] Especially the rather low number of micro-protocol modules we were able to implement so far limited the application spectrum.

on the "smoothness" of the fitness landscape. If e.g. local maxima are common, a higher probability of random exploration is advised, but the error threshold[264,265] imposes an upper limit for this value. Likewise, low mutation rates in Genetic Algorithms cause the population to stagnate, whereas too high settings causes the algorithm to degenerate into random search.

Dynamically at Run-time

Our implementation enables the evolution logic to be dynamically re-parametrised at run-time according to a simple heuristic based on the fitness of the best stack found so far, which is a common strategy for adapting e.g. the mutation rate of Genetic Algorithms.[119,323] As mentioned above, a high random component or large step size can improve the evolution speed, but reduce the evolution quality, i.e. cause the algorithm to "leap" around the vicinity of the optimum without ever reaching it. Lowering e.g. the mutation rate for once the achieved fitness is high enough might thus improve the performance of the algorithms. We utilise a very simple algorithm-dependent method for adapting the algorithms' parametrisation based on the fitness recorded for the best stack found so far, which is conceptionally identical for all algorithms. We therefore only present the adaptation formula for the Evolutionary Algorithm and the Composition Tree Search in here. While the results from our experiments with on-line adaptation of the Composition Tree Search's parametrisation are promising (see discussion in *Section 6.1.4*), we cannot give definite results of when and how to adapt the configuration just yet, and plan to perform further research in this direction in the future.

For the Composition Tree Search, we utilise the following formula to derive the algorithm's search size M_s based on the fitness of the best stack. Let $[\mathcal{P}_s^\blacktriangledown, \mathcal{P}_s^\blacktriangle]$ denote an user-defined range for M_s, with $\mathsf{f}_{best} = \max\limits_i \mathsf{F}[P_i]$, and $\mathsf{f}^\blacktriangledown, \mathsf{f}^\blacktriangle$ a user-configured fitness range. Then

$$M_s = \begin{cases} M - M_e - \mathcal{P}_s^\blacktriangle & \text{if } \mathsf{f}_{best} \geq \mathsf{f}^\blacktriangle \\ M - M_e - \mathcal{P}_s^\blacktriangledown & \text{if } \mathsf{f}_{best} \leq \mathsf{f}^\blacktriangledown \\ M - M_e - (\mathcal{P}_s^\blacktriangle - \mathcal{P}_s^\blacktriangledown) \dfrac{\mathsf{f}_{best} - \mathsf{f}^\blacktriangledown}{\mathsf{f}^\blacktriangle - \mathsf{f}^\blacktriangledown} & \text{otherwise,} \end{cases}$$

defines the derivation of M_s from the fitness. The local search distance δ already depends on the fitness of the previous generation, as described in *Section 4.1.3*.

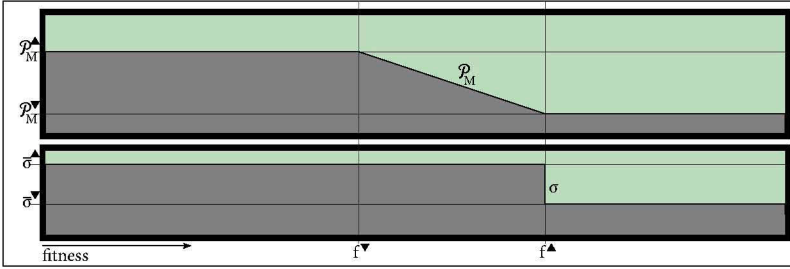


Figure 4.8: *The configuration of the Evolutionary Algorithm changes depending on the achieved fitness of the best stack configuration found so far: For higher fitness values, the need to make radical changes gets reduced, i.e. $\mathcal{P}_M$ and σ are set to lower values.*

For the Evolutionary Algorithm, we implemented the following formulae which set user-configurable ranges for the mutation probability and the variance of the normal-distributed random variable used for continuous control values (see also *Figure 4.8*), $[\mathcal{P}_M^{\blacktriangledown}, \mathcal{P}_M^{\blacktriangle}]$ $[\sigma^{\blacktriangledown}, \sigma^{\blacktriangle}]$.

$$\mathcal{P}_M = \begin{cases} \mathcal{P}_M^{\blacktriangledown} & \text{if } \mathsf{f}_{best} \geq \mathsf{f}^{\blacktriangle} \\ \mathcal{P}_M^{\blacktriangle} & \text{if } \mathsf{f}_{best} \leq \mathsf{f}^{\blacktriangledown} \\ \mathcal{P}_M^{\blacktriangle} + (\mathcal{P}_M^{\blacktriangledown} - \mathcal{P}_M^{\blacktriangle}) \dfrac{\mathsf{f}_{best} - \mathsf{f}^{\blacktriangledown}}{\mathsf{f}^{\blacktriangle} - \mathsf{f}^{\blacktriangledown}} & \text{otherwise} \end{cases}$$

$$\sigma = \begin{cases} \bar{\sigma}^{\blacktriangledown} = \max\left(\min\left(\mathcal{D}_k\right), \sigma^{\blacktriangledown}\right) & \text{if } \mathsf{f}_{best} \geq \mathsf{f}^{\blacktriangle} \\ \bar{\sigma}^{\blacktriangle} = \min\left(\max\left(\mathcal{D}_k\right), \sigma^{\blacktriangle}\right) & \text{otherwise} \end{cases}$$

For both algorithms we thus maximise the random element of the algorithms and make them more greedy when the fitness is low. For the Evolutionary Algorithm, this means increasing the mutation rate, for the Composition Tree Search, it involves increasing the probability of random experimentation and increasing the size of the area in which the local search is performed.

4.2 Mid-Term Decisions – Classification & Population Selection

Our methodology for mid-term adaptation is again inspired by nature: Physically isolated populations of animals, e.g. those located on different islands, independently adapt to the local environment and over time evolve

into different species adapted specifically to the island they live on. As we intend our system to adapt to different conditions encountered in the network, we modelled our design after this phenomenon, in a way which is very similar to Mori *et al.*'s environment identifying genetic algorithm [250]: In our system different situations, i.e. the on-going traffic and the network conditions, are distributed onto different islands. Our population selection mechanism maintains one distinct sets of stack blueprints for each island, which we call the population of that island. The situational classifier identifies the current situation based on sensor measurements, and then decides which population to use. It ensures that similar situations are grouped into the same population, i.e. the state of the network and the traffic conditions under which the stack blueprints evolved are sufficiently similar. The best stack blueprint present in the selected population is used for normal operational traffic, and the candidate stack blueprints for experimentation are also selected from this population. Our implementation includes two classifiers, the matrix-based and the k-means method described below.

4.2.1 Matrix-based Selection

The matrix-based selection method arranges the populations into and m-dimensional matrix, and applies a simple decision heuristic on each axis of this matrix. The definition language we use for run-time configuration of the heuristic is very simple, and fully defined by the following EBNF:

```
functions = function, { ";", function } ;
function = axis, range, "=", body ;
axis = digits ;
range = "[", digits, ",", digits, "]" ;
body = arg ;
arg = "(", arg, ")" | digits | sensor | arg, op, arg |
        arg, "?", arg, ":", arg ;
op = "+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" | "==" | "!=" ;
digits = digit, { digit } ;
sensor = alpha, { alpha | digit | "_" } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
alpha = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
        "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
        "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

axis refers to the axis of the matrix to which the function is applied. *range* specifies the definition range of the function's codomain. *body* is the function body composed of arithmetic operators ($+, -, \cdot, \div$), comparison operators ($<, >, \leq, \geq, =, \neq$) which map to $\{0, 1\}$, and if-then-else branches (*cond* ? *true-op* : *false-op*). *sensor* refers to a sensor identifier.

As an example, a mobile device might benefit from choosing a different population based on whether the device is using a wired or wireless connection or based on whether the measured signal-to-noise ratio (SNR) is above a specific threshold. As a simple two-dimensional example consider the following case where sensor pwrstate reports 1 if the device is running on battery power, and 0 otherwise, and sensor snr reports the measured SNR. We now define one formula for each axis in the matrix:

```
0[0,1] = pwrstate;
1[0,2] = (snr < 30) ? 0 : (snr < 50) ? 1 : 2;
```

Thus, for example, population $\langle 1,0\rangle$ is chosen whenever the device is on battery power and the SNR below 30, as illustrated by the following table.

	pwrstate $= 0$	pwrstate $= 1$
snr < 30	$\langle 0,0\rangle$	$\langle 1,0\rangle$
$30 \leq$ snr < 50	$\langle 0,1\rangle$	$\langle 1,1\rangle$
$50 \leq$ snr	$\langle 0,2\rangle$	$\langle 1,2\rangle$

4.2.2 k-Means Clustering Adapted for Population Selection

k-means[150] is a common partitioning algorithm which distributes the elements of a dataset S into k clusters C_i based on centroids, i.e. the centre point of the cluster which is defined by the mean of the objects in the cluster, as represented by the pseudo code for **function** PARTITION_KMEANS in *Algorithm 4.16.*

The algorithm aims to maximise the partitioning quality, i.e. to ensure that the elements of a cluster are similar to each other but dissimilar to elements of other clusters, by making the clusters as separate and as compact as possible. The within-cluster variation that is used to measure the quality is defined as the sum of the squared error between all elements of the cluster and the cluster's centroid c_i for distance metric Δ, i.e.

$$E = \sum_{i=1}^{k} \sum_{s \in C_i} \Delta(s, c_i)^2.$$

This problem is known to be NP-hard even for two clusters, but efficient greedy algorithms exist that limit the calculation effort in practice, especially considering that in our case the number of data points in the set is equal to the number of distinct situations encountered up to a specified cut-off value evaluated and therefore comparatively low.

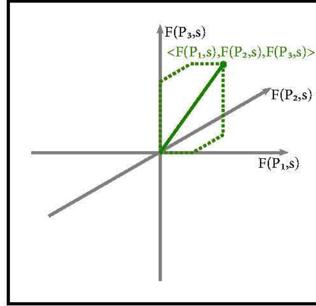


Figure 4.9: *The position vector used for clustering is constructed from the fitness of the baseline stacks $P_1, \ldots, P_n$ as measured for the situation s.*

Algorithm 4.16: k-Means Clustering

function PARTITION_KMEANS(Dataset S, Cluster Count k)
  $S' \leftarrow S$
  for $i \leftarrow 1 \rightarrow k$ **do** ▷ *randomly choose initial cluster centres c_i*
    $c_i \leftarrow \text{random}(S')$
    $S' \leftarrow S' \setminus \{c_i\}$
  end for
  $\delta_c \leftarrow \infty$
  while $\delta_c > 0$ **do** ▷ *repeatedly update clusters until nothing changes*
    $S' \leftarrow S$
    $C_i \leftarrow \emptyset$
    for all $s \in S'$ **do** ▷ *assign each object to the most similar cluster*
      $i \leftarrow \arg\min_i \Delta(s, c_i)$
      $C_i \leftarrow C_i \bigcup \{s\}$
    end for
    for $i \leftarrow 1 \rightarrow k$ **do** ▷ *update cluster centroids*
      $c_i \leftarrow \text{mean}(C_i)$
    end for
    $\delta_c \leftarrow \sum_i |C_i \bigcap C'_i|$
  end while
  return $\{\langle C_1, c_k\rangle, \ldots, \langle C_k, c_k\rangle\}$
end function

Our contribution here is the adaptation of the algorithm for the purpose

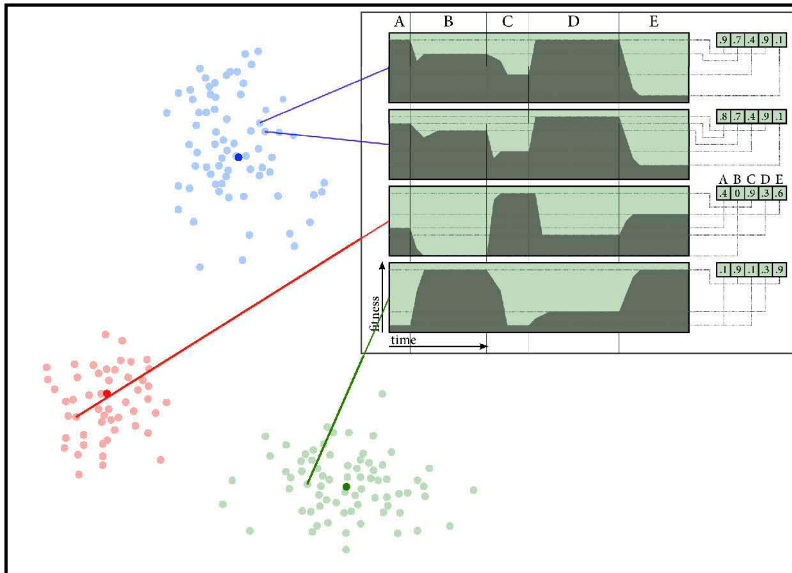


Figure 4.10: *A simplified illustration of our autonomous clustering method. The fitness measured for each of the baseline stack configurations serves as one dimension of a five-dimensional Euclidean space used for the cluster distance calculation.*

of population selection, and consists of the definition of the vector space and metric on which the algorithm operates. We wish to group situations into different clusters based on the effect they have on the stack operations, i.e. changes to the traffic and network conditions which do not impact stack operations shall be grouped in the same cluster, whereas changes that require a modification of the stack configuration to operate efficiently shall be distributed into distinct clusters. **We thus define the position of an object by the vector over the fitness values measured in the same situation for a set of distinct baseline stacks.** The metric we use for distance calculation is the Euclidean metric. The definition of the vector space is illustrated by *Figure 4.9*, and *Figure 4.10* shows a simplified version of the clustering algorithms, in which only two dimensions of the vector are actually represented, and the fitness is not normalised as explained in *Section 5.3.2*. While other classification criteria are possible, our definition has the important advantage that the effects of network conditions, etc., are already inherently included within the fitness value, and weighed according to their importance.

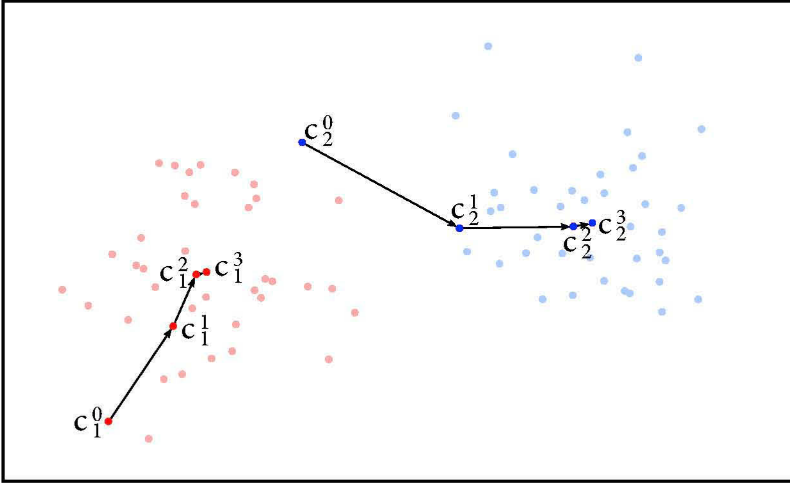


Figure 4.11: *The iterative centre update procedure of k-means. The initial, randomly chosen, centres of clusters C_1 (red), and $C2$ (blue) are represented by c_1^0, and c_2^0, respectively. c_i^j denotes their position after the j-th update.*

Thus no model or intrinsic knowledge of the impact of measurements on the fitness is needed.

Let $\dot{s} \in S$ denote the situation, i.e. the set of all current sensor measurements, P_i one of the n baseline stack configurations. We now define $\mathsf{F}(P_i, \dot{s})$ as the fitness of stack configuration P_i in this situation. The position vector s of situation $\dot{s}$ used for for clustering is now defined as

$$s = \langle \mathsf{F}(P_1, \dot{s}), \ldots, \mathsf{F}(P_n, \dot{s}) \rangle$$

For a discussion of the problem of how to select the baseline stacks, and how to deal with situations for which the baseline stacks' fitness is unknown, please refer to *Section 4.2.3*.

Figure 4.11 illustrates the iterative adaptation process of the cluster centres and the association of the data points with the clusters. The performance of the algorithm depends on the random choice of the initial cluster centres. To reduce the probability of bad random choices, the algorithm can be performed multiple times and the most appropriate result selected, e.g. based on the Bayesian Information Criterion.[185,237]

One limitation of k-means as currently implemented in our system is that the cluster count k has to be known in advance as it is configured at run-time.

We could work around this problem by providing an approximate range of possible k-values, applying the algorithm for each of these values and then choosing the best result, again by measuring the compactness and separation of the clusters. Faster alternatives, such as the X-means[277] algorithm, also exist. We plan to investigate other algorithms as well as automatic selection of k in the future.

4.2.3 Choosing Classification Criteria

The correct choice of classification criteria is critical for the effectiveness of our population selection method. If they are badly chosen, there is no measurable benefit, i.e. the best stacks evolved in two distinct populations will be identical. Such redundant populations reduce the adaptation speed, as the candidate stack configurations therein still have to be trialled on-line. The correct choice of criteria and the number of populations increases the overall system utility, as only then an optimal stack configuration that specialised for the situation can evolve. This effect is especially pronounced in the experiment described in *Section 6.2.1*.

Our current implementation requires an administrator to define these criteria by hand. But since our stack composition system should ideally operate fully autonomously, i.e. adapt the stack to the current situation with as little background knowledge as possible, this dependency on administrator is obviously unsatisfactory. We therefore began to investigate the possibility of truly autonomous selection of the classification criteria by the stack composition system itself. The k-means-based method we introduced above, can hopefully serve as a first step into this direction, as it classifies the situation solely based on the fitness of one or several baseline stacks, and thus needs neither a model of the world nor other background knowledge of the dependencies between sensor measurements and the resulting fitness. Two problems do however remain.

Firstly, we do not possess insight into how to select the baseline stacks. Currently we use a set of randomly generated stacks, as well as the unmodified IP stack, as baseline stacks, but further research is definitely needed. For example, imagine a scenario in which the only condition changing in the network is the bit-error-rate, and the fitness depends on the reception rate and the throughput. Here the IP stack's fitness can be considered a very representative measure to distinguish between different situations, as higher errors result in either a lower reception quality (in case of UDP traffic) or lower throughput (as TCP will retransmit lost packets). A stack which employs, for example, Hamming codes (see *Section B.2.2*) to compensate for transmission errors would however be assigned the same fitness value for every

situation in which the error rate is lower than the maximum it can compensate for, and its fitness should therefore not be used for classification in this case.

Secondly, we require a method for reliably handling situations for which the base stacks' fitness is unknown, as it is unlikely that we can procure samples of the performance of a set of baseline stacks for all possible sensor readings. Here we propose to let the system learn the weights for a metric over the sensor space, and select the closest known fitness value according to this metric. For example, if baseline stack P has only been tested under situations S' then we plan to use the following formula to derive the fitness for unknown situation s, where the weights ω_i are learned e.g. by an ANN or a naive Bayes classifier:

$$\mathsf{F}\left(P, \arg\min_{s' \in S'} \Delta_S(s, s')\right) = \mathsf{F}\left(P, \arg\min_{s' \in S'} \sqrt{\sum_i \omega_i |s_i - s'_i|^2}\right)$$

$$\omega_i = |S'|^{-1} \sum_{s' \in S'} \max_{1 \leq j \leq k} \mathcal{P}(s'_i | C_j),$$

where s'_i is the value of the i-th sensor in sample s' and $\mathcal{P}(s'_j | C_i)$ is the probability of sensor value s'_j appearing in cluster C_i. In other words, the more likely a specific sensor value is to cause classification into a specific cluster, the higher its weight.

4.3 Short-Term Decisions & Protocol Multiplexing

The methods for exchanging stack functionality described so far are insufficient if immediate action is required: If no error occurs, stacks are replaced only once their allotted time has elapsed, as described in *Section 5.3.2*. But in some situations choosing a different stack configuration based on, for example, the traffic type can be beneficial. For this purpose we provide the stack composition system with a means to encode run-time decisions into the stack.

4.3.1 In-stack Redirectors

Through the introduction of in-stack redirector module instances (see *Section 3.7.6*), the stack composition system can influence the control flow path within the stack. By introducing a redirector instance and assigning it as the destination of a connector in place of another module instance, the evolution logic and insert an instance analogue of a CPU `branch` instruction into

the stack, which forwards all calls to that connector to one of its connectors based on an arbitrary decision function. Since redirectors can be stacked, arbitrarily complex decision processes can be evolved by adding multiple, independently configured, redirector instances (see *Figure 4.12*).

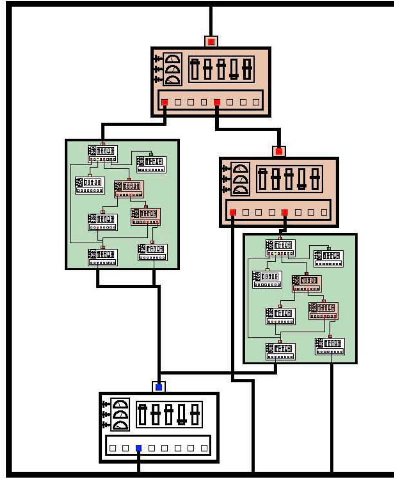


Figure 4.12: *Redirectors allow for the inclusion of complete or partial sub-stacks into another stack and can branch between them based on arbitrary decision criteria.*

If the maximum number of required control paths is known in advance, a static addition of an appropriate number of redirector and decision module instances is sufficient. But to fully utilise the possibilities offered by redirectors, an appropriate evolution logic which is capable of instance addition and removal of instances, has to be employed.

Redirectors can be specialised to provide any arbitrary service interface and provides N connectors bound to the same interface type, where N is the cardinality of the codomain of the decision function. The decision function can either be part by the implementation of the redirector module or provided by a separate decision module as described below.

Generic redirector modules are provided by the stack composition system for which, in the simplest case, the decision is based on whether the current value reported by a sensor is above or below a threshold. The sensor to use and the threshold value are defined through controls, so that the evolution logic can experiment with and decide on the correct setting to use.

Apart from these generic redirectors, specialised modules are provided which operate in a similar fashion to stack modules, but put additional requirements on the redirectors they provide. One such module we implemented decides for each packet whether to choose a reliable or transport protocol like TCP or rather utilise DCCP or UDP based on an arbitrary criterion such as destination, flow type, or the time of day.

4.3.2 Decision Modules

Decision modules provide a means to encode branching decisions for in-stack redirectors in a re-usable way and independent of their position within the stack. These modules themselves do not have access to the service interface or the associated data (e.g. data packets), but can read sensors and (and modify) the state information present in the persistent storage space described in *Section 3.4.1*. Thus decision modules are capable of offering two distinct application areas. Firstly, they can provide heuristics on which the branching is based, independent of the program flow. Thus the same decision can be applied at any position within the stack without requiring any changes to the implementation. Secondly, they are able to take the decision repeatedly and at different places in the stack, based on e.g. the current flow as reported by another module – for example be means of a sensor – that is located at a position in the stack where it has access to e.g. the necessary packet headers. Thus, for example, a decision module instance could return 1 whenever the link that a flow utilises reports a high error rate and 0 otherwise. Two distinct redirector module instances at different locations in the stack might then base their actions on the same decision and e.g. enable forward error correction or a reliable transport protocol. Decision modules are not limited to redirectors, but can be utilised by all stack module.

4.4 Summary & Conclusion

In this chapter we introduced our logic for autonomous adaptation, which consists of three interdependent layers: the long-term evolution logic, the mid-term situational classifier and population selector and the short-term in-stack redirector and decision modules.

We presented the evolution logics algorithms we designed and implemented in the stack composition system and explained why one single algorithm would be insufficient. We further described how these algorithms can be autonomously adapted at run-time. In the future we consider to explore this field further and to apply a meta-heurisitic, e.g. a learning algorithm to

autonomously select and adapt the evolution logic depending on the situation, in a similar manner as Fernandez-Prieto described for Genetic Algorithms,[111] as well as several other approaches.[30,100,102,201,320,321,322,378]

With respect to the mid-term adaptation layer, we discussed the two methods for autonomous population selection we implemented. While we cannot yet provide exhaustive and conclusive results as for how effective the different classification approaches are in a realistic environment, our experiments show the need for this or a similar approach: If the environment or traffic conditions change quicker than the adaptation process, sufficiently accurate evaluation and comparison of the fitness and thus the entire evolution process otherwise becomes impossible, as described in *Section 6.2.1*. Our current system still depends on administrator-configuration for the selection of classification criteria. In the future we plan to research how to further increase the autonomy of the system, i.e. how the stack composition system can learn these criteria on its own. For this purpose we presented an approach we intend to explore in the future.

Finally, we introduced our mechanism for short-term adaptation by means of micro-protocols placed into the stack by the evolution logic. While our previous experiments[170] with a similar approach look promising, they are based on a far more rudimentary system design, and not directly applicable to the current stack composition system. We therefore omitted a discussion of these results here, an intend to re-evaluate the approach within the current stack composition system further in the future.

Chapter 5

Stabilising the Measurement Environment

The stack composition system's ability for evolution depends on a reasonably accurate estimate of the stack configurations' fitness, i.e. the performance of the corresponding stacks. Most of the algorithms employed by the evolution engine use the fitness estimate to determine the direction of evolution, i.e. the likelihood of a stack to serve as basis for further exploration is proportional to its fitness. And since the stack composition system is supposed to determine the fitness experimentally, it has to extract the necessary information from the environment in a precise and reliable way.

The normal operations of the system likewise depend on situational awareness, as the system decides which stack to use based on its assessment of the situation as described in *Section 4.2*. It also needs to be aware of operational problems, execution errors, abysmal stack performance, etc., because it has to abort its experiments with the stacks that cause these problems right away.

In this chapter we focus on the problem of reliable gathering the necessary data, assessing the information contained therein, and deriving a fitness measure from this information. We begin by discussing the means by which the stack composition system can acquire the needed sensory data, then discuss the influence the varying conditions of the network can have on the information gathering process, and describe how the actions of the system itself can aggravate these problems. We then discuss how our system extracts the information contained within the noisy sensor data, how it can mediate some of the mentioned problems, and further elaborate on how different

aspects of our system are designed to counteract the mentioned problems.

5.1 Sensor Information

In our design sensors act as an abstraction for various types of state or environmental information. Sensors either serve as low-overhead state indicators (passive sensors) or collect sample data on request e.g. by collaborating nodes within the network (active sensors). Sensors are identified by system-local unique identifiers and used for various purposes by the fitness function, the stack steering system, and by stack modules. While the format of the sensor output may be proprietary, most sensors used in our implementation are integer-valued, and include a timestamp which indicates when the information was sampled and a confidence measure which indicates how reliable the reported value is expected to be, e.g. the mean squared error. As the access model of active and passive sensors differs, we discuss them separately.

5.1.1 Passive Sensors

Passive sensors are those sensors provided by the stack steering system and stack modules which provide access to readily available information. The stack steering system uses sensors to make available information gathered e.g. from the operating system or the network interface drivers. Similarly, stack modules use such sensors to expose state information – both locally-available and gathered from remote hosts – to other entities. Examples include the current bandwidth utilisation, cache hit ratio, or anything else that is considered useful for other modules, debugging, or fitness calculation. Local sensors follow a push-model, i.e. their value is updated whenever the exporting entity deems necessary. Thus querying local sensors does not induce any additional overhead apart from the access to a node-internal data structure and is basically instantaneous. The uses of these sensors are manifold, e.g. for cross-layer optimisation or fitness calculation: Many of our experiments introduced in *Chapter 6*, for example, include a measure of the communication overhead, which is derived by sampling passive sensors that report the total amount of outgoing data in bytes at the application and physical layer at the beginning and end of each sub-trial.

5.1.2 Active Sensors

Active sensors provide information the gathering of which requires actions to be performed that can have a detrimental effect on the system or network performance, for example because they are calculation-intensive or cause a

non-negligible amount of network traffic. These sensors are enabled on demand based on subscription and operate asynchronously: To access an active sensor, the requesting entity must subscribe to the sensor, which is identified by a globally-unique identifier of the entity, e.g. a remote node which hosts the sensor, and a sensor identifier local to that entity. Subscription includes a specification of the data that is be reported, e.g. the sampling frequency, based on which the affected entity can decide on whether to grant the request and how to configure its measurement equipment. The operation thereafter depends on the requested service and can involve e.g. reports periodically sent directly to the subscriber or in response to specific access requests. Active sensors are used to abstract functionality exported by existing technologies such as SNMP or remote measurement facilities, for example those discussed in *Section 2.4*.

Passive sensors of collaborating remote nodes which use the stack composition system can also be accessed by means of active sensors. This functionality uses the sidechannel interface introduced in *Section 3.7.4*, and either queries the sensors on demand, or accesses a locally cached version, if the information stored there is recent, e.g. because a broadcast update was received.

5.1.3 Sensor Requirements

Since sensors are part of the system they gather information about, the observer effect cannot be completely avoided, as even minor operations induce overhead, and might change the outcome of an experiment.[325] We therefore aim to minimise such effects by passively gathering the needed information whenever possible, for example, by performing inline measurements, i.e. we extract the measurement data from already available information such as the normal operational traffic or modify operations in such a way that performance is not affected. The ImTCP[210] algorithm, for example, re-schedules the transmission of normal TCP packets such that the available bandwidth can be estimated from the measured RTT and thus makes active measurements and their associated overhead unnecessary. Additionally we utilise periodic status messages and their acknowledgements, which are transmitted between collaborating stack composition system hosts, to measure RTT when TCP is not in use.

5.2 The Problem of Noise

Raw sampled sensor data on its own is often unfit for direct interpretation and requires further processing to extract the required information. One of

the reasons for this is the chaotic nature of the environment in which these sensors operate. As most networks route data that was either directly generated by a chaotic system, e.g. its users, or at least influenced by unpredictable physical effects, the networks' behaviour is far too complex to simulate properly.[276] Sensor data retrieved from the network thus often resembles a noisy, fluctuating, and multi-dimensional signal, as shown in *Figure 5.1*. Furthermore, it is often even not easy to decide what and how to measure, e.g. the measurement process itself might not sufficiently capture or even distort the signal. In this section we introduce some of the causes of problems and relate how they affect the reliability of the system.

The Network as a Chaotic System

As introduced above, the environment the stack composition system is exposed to cannot be assumed to remain sufficiently static. Just as an user might decide to access a certain network resource at any arbitrary time, a route may change or a link fail in the same unpredictable fashion. Yet the situations under which we test the candidate stacks need to be comparable. While our system cannot prevent these effects, it still has to be able to detect changes in the network and traffic conditions, decide whether they affect its operations, and take appropriate countermeasures to ensure that the results gained from experimentation are comparable. Without further precautions, the system might otherwise misinterpret changes, which in actuality are due to external influence, as being caused by the currently tested stack configuration.

Unpredictable Applications Both communication-related applications and network services running on the system naturally influence the network conditions and the performance of the system. An automatic software update, for example, might be scheduled to happen at a fixed time every few hours or whenever the system is perceived to be idle. Of course, we do not expect these applications to conform to the needs of the stack composition system regarding e.g. the reproducibility of experiments, in fact we actually intend them to be ignorant about them, as we discuss below. Instead our system has to cater to their needs, as the performance of the running application to a large part define the utility of the system and therefore the fitness of the stack.

Unpredictable Users Even though the stack composition system is supposed to operate autonomously, the influence of the "human factor" onto its performance should also not be underestimated: The goal of the adaptation

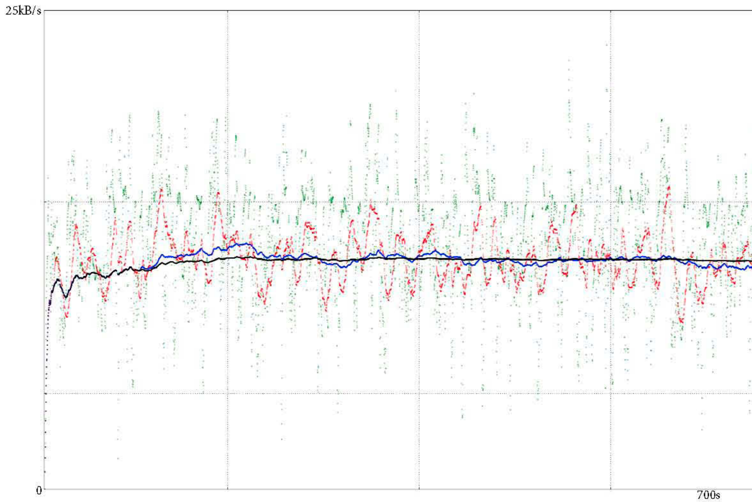


Figure 5.1: *This graph exemplifies the problem of and one countermeasure against noisy input signals. The green data points indicate the transfer rate measured per second at the recipient of a 12 kB/s bulk data transfer over a distance of 70 km on a busy VDSL link. The red, blue and black data points show the effect of averaging using a sliding window of 10 s, 100 s, and 1000 s, respectively. In this case rather primitive signal processing methods sufficed to transform the unusable input signal into an effective measure of the transfer speed. Depending on the measured quantity and the application requirements, however, the necessary normalization steps may be rather involved, especially, when the speed of change detection is vital.*

and therefore the evolutionary process itself is after all specified by the human users or administrators and has to be somehow conveyed to the system. Sadly, the users' understanding of the conditions that define the intended goal state is often vague, fuzzy, and sometimes even based on incorrect assumptions: It is, for example, often unclear how to evaluate how satisfied the users are with a provided service, and worse, how an autonomous system could possibly measure it. Most approaches therefore focus on expressing utility by means of easily measurable quantities[32,73,188,370] or use policy definitions,[141,168,337] which for many applications is definitely sufficient. Overall, a more explicit and direct approach for assessing the user preferences might provide more reliable results, but research in this direction has yet to

provide practical solution.[211,212]

An additional problem is presented by the unpredictability of the users' actions, and of the resulting effects on the measurements. In general, their behaviour becomes more predictable for larger "populations" and over longer periods of time: A single user might surf the web for an hour, but then suddenly initiate a large file transfer. This imposes a limit on the possible adaptation speed of the system, as it implies that more experiments need to be performed and a longer duration allocated for each experiment to smoothen spikes caused by sudden changes in behaviour.

The Influence of Experimentation

Experimentation itself also influences the conditions in the network, as different stack configurations produce e.g. different traffic patterns and thus can influence both future experiments and experiments performed at the same time on remote nodes. For example, even if an application were to send data at a constant rate, a defective stack configuration can cause the send queue of a socket or protocol to fill up and thus impose additional load on following experiments. The system therefore has to safeguard against such effects, either pro-actively by repairing the "damage" or passively by delaying follow-up experiments such that the likely impact is minimal. Measurement data can still exhibit effects caused by previous experiments, even after the action is long finished. Data aggregation or normalization approaches that utilise a sliding window approach can make matters worse and further extend this influence, as do transmission delays induced by the network. And as communication is not instantaneous, the network itself also causes delays in the reporting of measurement data.

In a distributed set-up, where every node can schedule its experiments independently, effects caused by other experimenting nodes can influence the fitness evaluation, and this influence is not limited to the trial time, as visualised in *Figure 5.2*. Experimentation can furthermore affect the actions of the users and application, which might, for example, adapt the codec used in a VoIP call to the available bandwidth or postpone watching a streaming video when the throughput is bad. And active measurements can further aggravate this problem.

Probing in the Dark

Worse than noisy data, however, is no data at all. The observability of the network environment is limited to the data provided by sensors it has access to, and the information that e.g. remote nodes are willing to share is

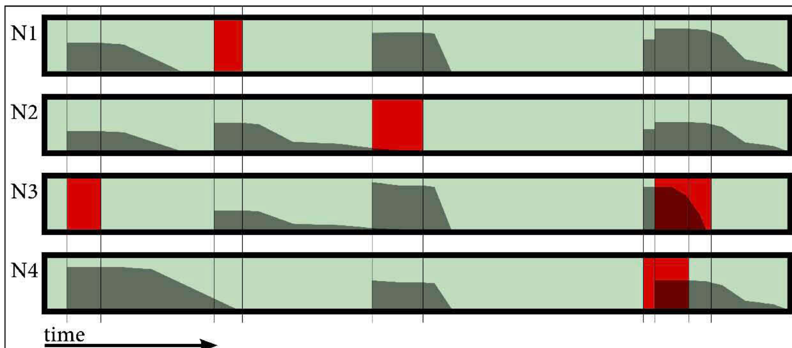


Figure 5.2: *Experiments can influence the operations of other nodes. In this 4-node example, the experimentation phase of a node is marked in red, the overall strength of fluctuations other nodes experience due to this experimentation is marked by grey graphs. The strength of these effects often varies between nodes, e.g. experiments of node N1 do not influence N4. Effects are usually especially pronounced when the experimentation phases of multiple nodes overlap, but can extend far beyond the actual experimentation interval.*

often constrained by policy and business interests, and also because packet-switched networks were not designed with measurements in mind.[365] The problem of lacking data is especially pronounced in our case, as remotely-gathered information still needs to be conveyed across the network, and an ineffective or buggy stack can seriously impede this process.

Possibly even worse than missing data is incorrect measurement data, as our system on its own cannot verify that the measurements are reliable. For example, when the fitness assessment is based on sensor information gathered from the stack modules themselves, these modules can – intentionally or due to programming errors – falsify the reported data and thus steer the evolution into an unwanted direction. An impressive example of this problem was reported for autonomously synthesised protocols, which simply generated perfect transmission statistics, but did not even try to fulfil their purpose.[387]

When reports from remote entities attribute to the fitness calculation, the inherent need to trust the measurements becomes particularly pronounced, as the local system is normally easier to control than the entire network. In a worst-case scenario, a knowledgeable and resourceful attacker can effectively undermine the adaptation process. Consider for example the rather simplistic case of an attacker with access to a web server, which the target of the

attack uses to measure e.g. the delay and throughput. The attacker performs a DoS attack whenever the attacked node tests a good stack configuration, and does nothing when the stack configuration is known to perform badly. It gains the necessary knowledge about the attacked node's stack remotely, for example, by inferring the configuration parameters of TCP.[175,268] If this attack is implemented correctly, the stack composition system will arrive at the conclusion that the bad stack configurations are more effective than the actually good ones, and the stack eventually evolve towards the global pessimum instead of the optimum. The factors that make this attack possible solely depend on the run-time system configuration, and which the stack composition system cannot influence.

Independent Optimization towards a Common Goal

A further interesting problem, which we consider out-of-scope for our current work but cannot leave unmentioned, is how to ensure that multiple nodes within the network, which autonomously try to optimise the stack, eventually arrive at a stable common global optimum. For example, consider the problem of evenly distributing the work load amongst multiple nodes in a decentralised manner, with the aim of minimising the processing delay and resource utilisation. Possible implementations include service deployment via mobile code or agents, as introduced in *Section 2.7.3*. Whereas the information needed as input to the local fitness functions can easily exchanged between the nodes, one major problem remains: The fitness functions needs to be formulated such that the Nash Equilibrium falls onto the global optimum, since our concept models a non-cooperative game between the collaborating nodes. And while this is sometimes possible, the adaptation speed will be polynomial in the number of nodes, and in the worst case rise exponentially, as has been shown for the convergence to a desirable equilibrium (see e.g. chapter 5 of the ANA[16] deliverable D3.8,[48] which provides a detailed introduction to the problem in the context of α-threshold congestion games).

Reformulated for our use case, the problem can be simplified as follows: If the local utility of a stack is not influenced by the remote stack configurations, then the optimisation behaviour will be unaffected, i.e. every node can operate as if it was the only one optimising its stack. If the relative order of stacks w.r.t. their fitness is not influenced by the stack configurations used by remote nodes, then the normal actions we deploy to counteract situational changes are sufficient. If however a locally deployed stack performs better than another one if a specific remote stack is deployed, and the opposite is true for a different remote stack configuration, then the optimisation process

will take exponentially longer, or become impossible altogether.

5.3 Stabilising the Measurement Environment

In this section we describe the measures we implemented in the stack composition system that enable it to extract the information it needs from the raw sampled measurements and describe how the help to tackle the aforementioned problems. Our approach for performance assessment was inspired by Ramos-Munoz *et al.*,[296] i.e. the stack composition system also performs trial-and-error experimentation on network stacks, and utilises a fitness function based on measurement data from the network. We further realised and considerably extended some of their proposals for future work, as they for example only operated on individual protocols and suggested the use of an Evolutionary Algorithm as adaptation logic. Similarly to Bicket,[35] and as opposed to e.g. ADROIT,[357] we implement a heuristic that limits experimentation to a subset of the possible stacks i.e. in our system only those stack that the evolution logic considers likely to exhibit higher utility are actually evaluated. Additionally we abort stack trials whenever the conditions change considerably, experimental results are non-representative, or a problem is noticed during execution. Our methodology for measuring and experimentation is mainly based on standard normalisation techniques, such as averaging over multiple samples, elimination of outliers or anomalies, and fault-detection.

5.3.1 Measurements & Data Processing

During its operation the stack steering system constantly gathers measurement data from sensors within and outside the system, processes it, and acts on the resulting information. All sensor measurements and messages are timestamped, which enables the stack composition system to derive a sequential order of events and in specific cases to correlate causality. The system clocks of all nodes are synchronised by means of NTP and thus achieve an accuracy in the 10 ms range,[241] which for our purposes is more than sufficient. To compensate for network-induced delays, we process the sampled data and calculate the fitness only once a "calm-down" period is complete, which we discuss in *Section 5.3.2.* All reports are transferred over the reliable transport channel, i.e. lost report packets are retransmitted, and reports are cryptographically signed to reduce the risk of (in-)voluntary data corruption. We further took care to – whenever possible – utilise robust statistical estimators, i.e. we trim outliers from the sampled data and average over a sufficiently large number of samples to reduce the effect of sudden spikes,

like those illustrated in *Figure 5.1*. Sensor measurements are weighed according to their confidence value, which is part of our sensors, and which further helps to increase the accuracy of the aggregated information.

5.3.2 Experimentation

All stack configurations are evaluated by means of experiments that the stack steering system schedules based on its assessment of the current situation. It ensures that the conditions of trials are comparable by means of the following measures, which we discuss in more detail below.

- Every stack is tested multiple times in independent sub-trials.
- The order of these experiments is randomised.
- After each experiment a "calm-down" phase is enforced.
- Experiments are performed infrequently and after random intervals. The length of these intervals is proportional to the system performance.
- Unsuitable stacks are prematurely terminated.
- Results of experiments are only compared with those achieved under sufficiently similar situations.

Trials and Sub-trials

Due to the noisy nature of the experimentation environment, one single sample, i.e. one experiment with a particular stack configuration, is often not sufficient to reliably assess the result. The stack composition system therefore performs multiple independent experiments, which we call sub-trials, and averages the results as discussed below. A trial is the sum of all sub-trials performed for a particular stack configuration.

Number of Sub-trials to Perform

The decision of how many sub-trials to perform for every stack configuration that is being trialled influences the precision and speed of the adaptation process: Additional experimentation naturally takes time and thus slows down

the evolution. It is however vital to perform a sufficiently large number[41] of independent experiments, i.e. sub-trials, as like with every stochastic process, the accuracy of the measurements increases – up to a certain point – with the number of samples obtained. Assuming that the conditions during the experiments are randomly distributed, we average over the achieved fitness values per stack configuration, and thus hope to achieve a utility measure which is representative for the situation under which the experiment was performed. As the accuracy of the fitness estimate itself also varies – depending on the network and traffic conditions – our averaging process includes a measure of confidence per sub-trial, based on the sensor confidence measure described above. Let c_i denote the confidence of fitness measurement f_i, then the average fitness is calculated as

$$\mathsf{F} = \frac{\sum_i \mathsf{c}_i \mathsf{f}_i}{\sum_i \mathsf{c}_i}.$$

The number of sub-trials to perform has to be configured by the administrator. A dynamic determination of the number of samples needed, e.g. by means of the variance, would require knowledge about the statistical distribution of the sample values. The stack composition system does not possess such knowledge, as the fitness function is defined at run-time. The minimum number of sub-trials performed is limited to twice the population size. We randomise the order in which stack configurations are trialled, so that the effects stacks can have on consequent experiments are minimised.

Experimentation Length per Sub-trial

For a correct assessment of the fitness, it is vital to allocate enough time to the individual experiments. Each experiment needs to be long enough for the protocol behaviour and traffic conditions to stabilise. But allocating too much time for each sub-trial is also counter-productive, since it effectively increases the time during which potentially ineffective stacks are deployed and thus reduces the overall system utility.

The length of a sub-trial itself is variable and depends on the deployed micro-protocol modules, as many protocols need a specific time to stabilise. The minimum time Δ_m needed for module m is therefore included in the

[41] For our experiments on a rather busy VDSL link, we empirically found 5 to 10 sub-trials were sufficient to normalise the measurements. These results, however, depend largely on the experimentation environment, and cannot be generalised without performing an exhaustive evaluation of all possible network environments, which we deem out of scope.

module specification introduced in *Section 3.4.1*. The fitness function specification also includes a minimum time Δ_f for this purpose. The minimum length of a sub-trial is consequently defined as the maximum over Δ_f and the Δ_{m_i} of all modules included in the stack. If during this period any unforeseen problems occur (see *Section 5.3.2* below), the execution is aborted prematurely. Otherwise, after the minimum trial time has elapsed, the fitness function is polled. If the returned confidence estimate c is above a pre-defined threshold c_1, the test is complete. Otherwise, additional time in steps of Δ_1 microseconds can be allotted up to a pre-defined maximum of Δ_{max}, depending on the fitness in relation to c. The pseudo code in *Algorithm 5.1* illustrates the process.

Algorithm 5.1: Determining the length of a sub-trial

$t_0 \leftarrow$ current_time()
start_stack(S)
$\Delta_0 \leftarrow \max\left(\Delta_{min}, \min\left(\{\Delta_m \mid \forall m \in S\} \cup \{\Delta_f\}\right)\right)$
wait(Δ_0)
if stack error **then**
  return $F_0 \leftarrow 0$
end if
while current_time() $- t_0 < \Delta_{max}$ **do**
  $F_0, c \leftarrow$ evaluate_stack_performance(S)
  if $c \geq c_1$ **then** ▷ *if confidence is high enough, accept*
    return F_0
  else if $c \geq c_0 \wedge F_0 < f_0$ **then** ▷ *if confidence is reasonably high and*
  ▷ *the fitness appears to be bad, abort*
    return $F_0 \leftarrow 0$
  end if ▷ *otherwise continue to test the stack, up to a maximum*
  wait(Δ_1)
end while

Delay between Experiments

After an experiment is complete, the stack composition system switches to a baseline stack, i.e. a known-good stack[3], and uses this stack for a pre-defined time period. Through this measure we hope to compensate for the possible

[3] This can be either a generic networks stack or the best stack found so far through experimentation.

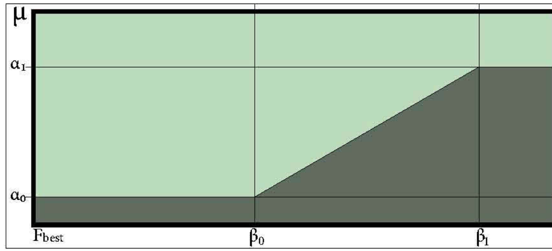


Figure 5.3: *The probability of a scheduling an experiment depends on the fitness of the best stack found so far.*

destabilising effects that the candidate stack can have on the conditions in the network. After this period the system monitors several indicators, such as the amount of data present in the application send queue, to determine whether the situation has returned to a stable state. As long as this is not the case, no further experiments are performed.

Once the situation has normalised, the next sub-trial is scheduled after a random delay that is several times larger than the actual experimentation phase. We thus try to ensure that experiments are infrequent and that overlaps between the experimentation phases of different nodes become unlikely.

The delay is calculated by means of a Gaussian variable, the mean of which depends on the fitness of the best stack found so far within the current population, and thus the current situation as discussed in *Section 4.2*. The system thus biases towards exploration when the fitness under the current conditions is low and towards exploitation when it is high, as we assume that a higher fitness reduces the need to find a better stack. For this purpose we let the administrator define two floating-point ranges, $[\alpha_0, \alpha_1]$ and $[\beta_0, \beta_1]$. α_0 defines the minimum delay between two experiments, and α_1 the respective maximum. The β_i in turn define the fitness values for which the minimum or maximum delay is chosen, respectively. If the fitness lies between the β_i, we linearly[☞] interpolate between α_0 and α_1, as illustrated by *Figure 5.3*. The mean of the Gaussian variable is determined by the following formula, in which F_{best} is the fitness of the best stack in the selected population, and μ

[☞] This choice was arbitrary. For example, using a sigmoid function would also have been possible. Determining the most effective function for this purpose is left for future work, as we currently cannot reliable estimate the influence of this function on the overall system performance.

the mean of the Gaussian variable defined as

$$\mu = \begin{cases} \alpha_0 & \text{if } \mathsf{F}_{best} \leq \beta_0, \\ \alpha_1 & \text{if } \mathsf{F}_{best} \geq \beta_1, \\ \alpha_0 + (\alpha_1 - \alpha_0)\dfrac{\mathsf{F}_{best} - \beta_0}{\beta_1 - \beta_0} & \text{otherwise} \end{cases}$$

We further define the variance relative to the mean as $\sigma^2 = \frac{\mu}{4}$.

Premature Trial Termination

If stacks perform badly, their execution is aborted before the allotted trial time has expired. Causes for such premature termination are a particularly low in-trial fitness estimate as described above, as well as the reasons introduced below. Whenever a trial is terminated, the stack configuration is assigned the minimum possible fitness value.

Termination of Abysmal Stacks Stack execution is terminated by the stack steering system as soon as it detects that the tasks assigned to the stack are not being performed. If e.g. an application tries to send data through the stack, but no outgoing packets arrive at the physical layer within a reasonable time, this condition is assumed. If the candidate stack is unable to communicate with a node that is reachable by other means – for example, via the sidechannel interface introduced in *Section 3.7.4* – the stack is immediately aborted. To reduce the effect of false positives, even aborted stacks are tried at least trice.

Termination due to Complaints Even if the node itself seems to be able to operate reliably, an immediate termination of execution is sometimes warranted. If a faulty protocol were to flood the network with messages, execute a DoS attack, or otherwise impede the correct operations of the network, the affected nodes can send a complaint via the reliable sidechannel described in *Section 3.7.4*). Depending on the trust-relationship with the node that sent the complaint (see *Section 5.3.4*), the stack configuration is either immediately purged, or retried at a later time as described above. The reasoning here is that if, for example, the only router on a subnet sends such a complaint, the possible effect of being separated from the network by this router outweighs the effect of possible false positives.

Termination of Invalid Stacks Invalid stack configurations that are not detected at composition time will be terminated in all of the following situations.

Loop Detection The stack steering system keeps track of calls to all connectors in the stack: Connectors are "locked" when they are invoked, and "unlocked" once the call returns. If the lock count of any connector exceeds the defined maximum (three in the current implementation), an infinite loop is assumed, and the stack therefore terminated immediately.

Error Handling When code execution errors, such as C++ exceptions or segmentation faults, are detected, the stack is also immediately aborted. Regrettably, but obviously, not all such errors are detectable or recoverable, as for example memory corruption bugs can leave the entire system in an unstable state. Solutions and safeguards for this problem are again considered out of scope for this thesis.

5.3.3 Ensuring Comparability

We use the classification methodology introduced in *Section 4.2* to guarantee that the situations in which the experiments are performed is sufficiently similar. The stack composition system performs the classification process periodically, and at a higher frequency than the actual sub-trial length. Whenever the classification changes, i.e. a different population is tested, the current trial is aborted, the results discarded, and the baseline stack used to guarantee that the system returns to a stable state.

Our reason for using only this technique is simple: The goal is after all to let the evolution logic adapt to the entire spectrum of situations covered by the classification process, i.e. to handle all possible conditions therein. Ideally the difference in fitness achieved by the same stack configuration for different situations within the same cluster should be marginal. Otherwise, the number of performed sub-trials must be sufficiently high to maximise the probability of all stack configurations being tested in a representative subset of situations in accordance to the underlying statistical distribution. Since the fitness function and classification parameters are configured at run-time, this responsibility fundamentally lies with the administrator. Whether our approach for autonomous classification based on the fitness difference of known-good stacks (see *Section 4.2.2*) is helpful for this purpose requires further evaluation in the future.

5.3.4 Information Exchange

To increase its situational awareness, the stack composition system depends on information from collaborating entities within and outside the local node. In this section we introduce the provided facilities through which the system can interact with remote nodes and the applications running on top of it.

Trust relationship

The stack composition system builds a trust-relationship with remote entities to guard against attacks. For this purpose the stack composition system uses trust levels expressed as floating-point values in the range $[0, 1]$. The administrator can configure a minimum trust threshold, and assign public keys and trust levels to remote nodes and applications, and specify the maximum number of hops for propagating trust, as well as a discount factor.

The system learns the trust levels and public keys of initially unknown nodes from those nodes it already knows by means of polling. If several trusted nodes assign different trust level, the level reported by the most-trusted already known node is used. For equally trusted nodes, the minimum reported value is used. The assigned trust level is the product of the trust level for the directly known node, the reported trust level of the unknown node, and the discount factor. Since nodes communicate over the sidechannel, and all data transmitted over this interface is cryptographically signed, we assume that reports cannot be falsified by third-parties.

The Concept of Entity Satisfaction

Similarly to the scalar fitness measure we use to assess the utility of a particular stack configuration, we employ a scalar measure to express how content other entities, e.g. nodes in the network, applications or users, are with the current situation and the actions of the stack composition system. Consequently, we call this measure the entity satisfaction.

We define satisfaction as a floating-point value in the range $[-1, 1]$, where 1 indicates perfect operation conditions, i.e. content with the current state, -1 total discontent, and 0 indifference. Trusted entities can report two types of satisfaction to the stack composition system through a special communication interface: The global satisfaction expresses the content with the overall situation, and the specific satisfaction refers to the (dis-)satisfaction with the behaviour of a particular node in the network.

Similarly to the trust values defined above, every known entity is attributed a configurable weight which defines the significance of the entity to the local system. The system primarily uses these reports to determine

whether to prematurely terminate an experiment, but these reports can also be included into the fitness function, e.g. when the aim is to reach a shared global optimum for multiple nodes (see *Section 5.2*).

Performance Notification Interface

The stack composition system provides a common interface through which applications and remote nodes can report their satisfaction with its actions and the global situation in general. They report distinct satisfaction values for both cases either via the sidechannel interface in case of remote nodes, or using IPC in case of applications. Just as for sensors, satisfaction values also include a measure of confidence, which denotes the presumed accuracy of the report.

Notification by means of Broadcast Messages The general satisfaction is broadcast to all trusted nodes in the network neighbourhood. Node-specific satisfaction values are only broadcast if the specific effect of a node exceeds a defined threshold: If the influence difference between a node and the average is low, the general satisfaction is sufficient and the traffic overhead is kept minimal. If one node's actions have a particularly strong effect – for example, if it is (maybe unwittingly) performing a DoS attack on another node – this information is immediately conveyed to all other nodes.

Notification to Specific Nodes Specific nodes are informed about their satisfaction only if separate per-node measurements are available and the corresponding node-specific satisfaction is sufficiently different from the general satisfaction. Such reports are transported via single-cast and used to terminate experiments with faulty stacks – provided that the reporting node's trust level is high enough.

Notification from Applications At start-up local applications register with the system and set the initial satisfaction value, and the associated confidence estimate. Every application is additionally assigned a weight by the administrator. The application satisfaction is defined as the weighted sum of the individual applications' reports and used for the same purposes as remote nodes' reports – to terminate faulty or inefficient stacks and for fitness calculation.

5.3.5 Fitness Calculation

For the stack composition system, the most important measure that can be extracted from the sensor data is the fitness, as it encodes the presumed utility of the stack. Consequently it directs the long-term evolution as described in *Section 4.1* and even serves as a measure of the distance between situations in the autonomic classification process detailed in *Section 4.2*.

The task of extracting the fitness from the sensor measurements is attributed to the fitness function, which is probably the most common[⊑] approach for this purpose, and which is defined at run-time by the administrator, and expressed by means of the following EBNF:

```
functions = trialtime, function, { function } ;
function = funcname, "(", paramname, { ",", paramname }, ")" , "=",
           expr, ";" ;
expr = "(", expr, ")" | float | inputname | paramname | expr, op, expr |
       funcname, "(", params, ")" | expr, "?", expr, ":", expr ;
op = "+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" | "==" | "!=" ;
params = expr, { "," expr };
paramname = alpha, { alpha | digit | "_" } ;
funcname = alpha, { alpha | digit | "_" } ;
inputname = alpha, { alpha | digit | "_" } ;
trialtime = "[", digits, "]" ;
float = digit, { digit }, ".", { digit };
digits = digit, { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
alpha = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
        "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
        "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

trialtime is the minimum time in microseconds to allocate for one subtrial. *funcname* denotes either an explicitly defined function, fitness, or one of the following pre-defined functions:

$$\begin{aligned} \text{power}(x, y) &= x^y \\ \exp(x) &= e^x \\ \log(x, y) &= \log_y x \end{aligned}$$

[⊑] See *Section 2.5.2* for several use cases explored by others.

$$\begin{aligned}
\mathrm{erf}(x) &= \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,\mathrm{d}t \\
\text{apply_bias}(x, \beta) &= \mathsf{b}_\beta(x) = (1 - \beta) + \beta x \\
\text{saturate}(x, l, h) &= \begin{cases} l & \text{if } x < l \\ h & \text{if } x > h \\ x & \text{otherwise} \end{cases}
\end{aligned}$$

The name fitness denotes the function that is called to calculate the fitness. *inputname* refers to one of the inputs discussed below.

Input Selection

The fitness function can base its calculations on the following input data.

Sensor Measurements Both active and passive sensors, as introduced in *Section 5.1* are available as input. The fitness assessments fed into the evolution logic are derived based on data gathered during each experiment through the application of pre-defined functions, for example, to derive the mean or maximum over all gathered samples, or to access the first or last sample.

Application & User Satisfaction Apart from the physically measurable quantities introduced above, our concept also explicitly allows for a more dynamic and flexible approach: When the administrator does not possess sufficient knowledge to express the utility by means of such quantities, they can delegate this task to those entities that do, i.e. the applications and / or users. Here app_sat(app) denotes the satisfaction reported by a specific application, and app_sat() is the weighed sum over the reported fitness of all registered applications, i.e. F_A is calculated using $\mathsf{F}_A = \frac{\sum \mathsf{S}_i \omega_i}{\sum \omega_i}$, where S_i is the satisfaction of application i, and ω_i the associated weighing factor, which is defined by the administrator of the stack composition system.

The purpose of this approach is to leverage application-level knowledge, which cannot easily be accessed by the stack composition system. After all, the application itself is in a far better position to assess its communication requirements than an external entity ever could. As an example, an FTP client could define its satisfaction as $\mathsf{S}_1 = \bar{T}$ if a file was transmitted correctly, and $\mathsf{S}_1 = -1$ otherwise, while a VoIP phone might prefer to report $\mathsf{S}_2 = \bar{T}\bar{D}^{-1}\bar{J}^{-1}$, where $\bar{T}$, $\bar{D}$, $\bar{J}$ are normalised measures of the throughput,

delay, and jitter, respectively. While the experiments we describe in *Chapter 6* do not use this facility, it has already been extensively utilised to measure the satisfaction for bulk transfer, video streaming and VoIP applications and derived the fitness estimate from them.[192]

Collaborative Adaptation For collaborative adaptation towards a common goal shared among multiple nodes, as introduced in *Section 5.2*, the fitness function can incorporate remote_sat(), which denotes the weighted sum over all satisfaction values reported by remote nodes, and remote_sat(node), which is the unweighed satisfaction reported by a specific remote node. If specific reports for this node are present, these are used, otherwise the general satisfaction as extracted from received broadcast reports is used.

Input Aggregation

Whereas the inputs can be processed and combined in any way representable by the EBNF given above, e.g. by means of the Cobb-Douglas production function,[89] we so far mostly utilised the following two approaches for weighing and balancing the effect of the different quantities. The first method is the common weighed sum, i.e. $\mathsf{F} = \sum_i \omega_i q_i$, where ω_i is the associated weight of quantity q_i, and which is closely related to the Von Neumann–Morgenstern utility theorem.[260] The second, multiplicative method, which we use for the experiments described in *Chapter 6*, is defined as $\mathsf{F} = \prod_i b_{\omega_i}(q_i) = \prod_i ((1 - \omega_i) + \omega_i q_i)$. We prefer this definition as it is easier this way to guarantee a minimal fitness value if only one condition is not met, i.e. $\exists i : q_i = 0 \iff \mathsf{F} = 0$.

Confidence

All sensor and satisfaction values that serve as input to the fitness calculation have an associated confidence estimate, which defines how reliable the input is supposed to be. For information reported by remote nodes that are not trusted the confidence is further reduced, i.e. all satisfaction and sensor measurements attributed to such a node are also distrusted. The confidence of the fitness assessment is calculated by replacing the input quantities, e.g. sensor data or satisfaction values, in the formula that represents the fitness function by the respective quantity's confidence.

Example

After the theoretical discussion above, we now present an example of how the fitness can be calculated from easily measurable sensor data. For this ex-

ample we assume that the goal is to maximise the achieved throughput, while also limiting the protocol overhead and guaranteeing reliable transmission.

Some easily measurable quantities can be used for fitness calculation without much preprocessing, e.g. the reception quality can be directly calculated as the ratio of acknowledged versus sent application-layer bytes:

$$\bar{R} = \frac{A_A}{S_A}$$

Since the information about the network environment is rather limited, we often need to use relative measures instead of absolute ones whenever possible. For example, the end-to-end bandwidth is not easy to determine passively, so we use the following function to achieve a measure in the range of $[0, 1]$ from the raw throughput T in bytes:

$$\bar{T} = 1 - 2^{-log_{10}\left(1+10^{-4}T\right)}$$

As we likewise do not know how much overhead will be considered optimal for future protocols, we use the sigmoid Gaussian error function to derive a scale-independent measure from the number of bytes sent on the application S_A and physical layer S_P:

$$\bar{O} = \text{erf}\left(\frac{S_A}{S_P}\right).$$

The aggregated fitness can then be defined directly based on the weights one wishes to attribute to a quantity, e.g.

$$\mathsf{F} = \bar{R} \cdot \mathsf{b}_{0.5}(\bar{T}) \cdot \mathsf{b}_{0.3}\bar{O}$$

would value correct reception higher than high throughput, and this again higher than low overhead. The confidence estimate c_F consequently would also be defined as

$$\begin{aligned}
\mathsf{c}_\mathsf{F} &= \mathsf{c}_{\bar{R}} \cdot \mathsf{b}_{0.5}(\mathsf{c}_{\bar{T}}) \cdot \mathsf{b}_{0.3}\mathsf{c}_{\bar{O}} \\
\mathsf{c}_{\bar{R}} &= \frac{\mathsf{c}_{A_A}}{\mathsf{c}_{S_A}} \\
\mathsf{c}_{\bar{T}} &= 1 - 2^{-log_{10}\left(1+10^{-4}\mathsf{c}_T\right)} \\
\mathsf{c}_{\bar{O}} &= \text{erf}\left(\frac{\mathsf{c}_{S_A}}{\mathsf{c}_{S_P}}\right).
\end{aligned}$$

5.4 Chapter Summary

In this chapter we introduced the problems related to the autonomous gathering of measurement data within a network setting, and its "distillation" into a easily processable form for fitness assessment and situational classification. We presented our design for performing experiments, acquiring measurement data, and extracting the information contained therein. We further described the countermeasures present in our framework for safeguarding against the uncertainties of the network. We also discussed the process of aggregating information into a scalar fitness value, and how the system can become aware of the needs and expectations of its applications and users.

Chapter 6

Experimental Validation

In this chapter we present some of the experiments we performed during our research. We focus on the feasibility of an stand-alone system for autonomous stack evolution within an actual network setting, i.e. are interested to find out whether a generic design such as ours is actually able to perform this task. We did not design our architecture with a specific application scenario in mind. It can in theory be applied to optimise the network stack to almost arbitrary conditions, definitions of utility, and operate on micro-protocols and service modules that are unknown at design or deployment time. But naturally the problem space here is huge and the uncertainties numerous[1], thus we were unable to explore the entire problem space exhaustively in our experiments, and instead decided to see whether our system as a whole is capable of adapting the network stack to the environmental and traffic conditions and thus increase the fitness of the system as a whole, even if only for a few specific test cases.

We implemented our entire framework both as a replacement for the network stack in the ns-3[1] simulator and as a stand-alone user-space application deployed in a physical test-bed, as described in *Appendix B.1*. One major problem related to this approach is the need for actual protocol implementations that can be composed and configured by our framework – a feat that most existing stacks were not designed for – and thus we had to (re-)implement all protocols we used for the experiments again on our own, which reduced the variety of scenarios we were able to test. However, we decomposed into configurable micro-protocol modules and implemented a

[1] For example, we do not know enough about the protocols that might be deployed on the system to make assumptions about their influence on our system's performance.

sufficient subset of the Internet stack, e.g. TCP, IP, and several additional protocols, as detailed in *Appendix B.2*, and used these for our experiments.

We begin by exploring how the evolution engine – which constitutes the long-term adaptation layer – fares when exposed to different adaptation scenarios. In *Section 6.1* we look at the adaptation dynamics of some of the algorithms we implemented in the evolution engine, to see how well the long-term evolution logic is able adapt the stack composition and configuration to improve the utility as measured by the fitness function. For this purpose we expose the system to different traffic scenarios and environments, both within a physical test-bed and in simulation. Here we utilise the physical test-bed only to validate that the results found through simulations reflect reality, as the overhead of actual physical experimentation precludes its use for a more thorough assessment. We aim to keep the networking and traffic conditions as realistic as possible, but focus on a single node running the stack composition system to eliminate possible side-effects caused by the interaction of multiple concurrently evolving systems, a problem we discuss in *Section 5.2*.

We then explore the capabilities of the mid-term adaptation layer, i.e. the performance of the situational classification and population selection approach, in *Section 6.2*. Here we investigate the usefulness of the matrix-based selection method introduced in *Section 4.2.1* by measuring its effect on the overall adaptation quality and speed in comparison to a system which tries to evolve one generic stack adequate for all possible situations.

Through these experiments we hope to provide first insights into the possibilities of autonomous stack evolution and to show its feasibility for some problem scenarios. While we do not exhaustively evaluate all possible aspects for adaptation, we do hope to lay the basis for future exploration.

6.1 Evaluation of the Long-Term Stack Evolution Approach

The long-term evolution approach encompassed in the evolution engine provides the mainstay of our concept's adaptation abilities, on which both the short- and medium-term functionalities depend. We therefore put the emphasis of our experiments on the validation of this approach, and begin by presenting a proof-of-concept for the system's ability to find a valid stack composition, followed by several scenarios to proof that it is additionally capable of finding a close-to-optimal configuration of the network stack's modules, both within a limited amount of time, followed by scenario designed to emphasize the differences in the adaptation characteristics of

the individual evolution logics. Lastly we investigate whether the performance characteristics of the different algorithms depend on the application scenario, and how the selection and parametrisation of the evolution logic influences the adaptation behaviour and speed.

6.1.1 Scenario 1A: Composition of the Internet Protocol Stack – Simulation

In this first experiment we provide a proof-of-concept of the evolution engine's ability to construct a valid stack composition out of a set of protocol modules. In particular, we intend to show that, given a limited but reasonable amount of time, the combination of on-line experimentation and creation of a follow-up generation through the evolution logic leads to a close-to-optimal stack configuration.

To prove that the stack composition system's ability to evolve the stack is independent of how the initial stack is configured, we let the system start from a set of randomised configurations, so that the initial stacks are with high likelihood either completely invalid, i.e. inoperable, or valid, but sub-optimal. We provide a fitness function which encodes our notion of optimality, set up a test-bed and expose the system to live application traffic. After each generation has been evaluated, we use the calculated fitness to assess the performance achieved by the stack that the system considers best at that point in time.

Our criterion for success is whether this stack performs close to optimally, according to the following definition. Since we cannot determine the true optimum without exploring all possible stack configurations, which is precluded by the size of the search space, we here define optimality as the fitness exhibited by the overall best stack found during the entire experimentation with this scenario. The initial generation of stacks is always randomly generated, and we performed more than 500 independent simulation experiment runs altogether, as described below. While such statistical sampling is not sufficient to determine the overall maximum, we consider the proximity to this value a considerable better measure of a non-deterministic evolution logic's capabilities for stack adaptation than simply measuring the improvement of fitness over time.

Adaptation Scenario

As scenario for our proof-of-concept we decided on the following question: Can the system compose the Internet stack? Or more precisely, given as input a set of networking protocols, i.e. variants of IPv4, TCP, etc., which

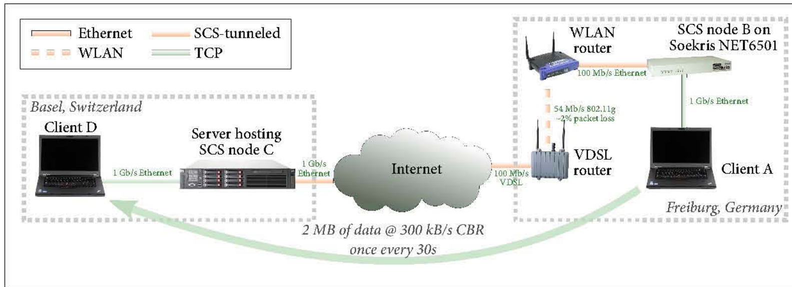


Figure 6.1: *The physical layout of our experimentation network is spread across two sites connected via the Internet. The simulation models the network conditions we empirically measured in this test-bed.*

are able to be arbitrarily stacked and which the stack composition system can combine in any way it decides, is the system able to come up with a composition that works over the Internet, and as a further goal, one that offers close-to-optimal fitness?

Network Topology We performed our experiment in a simulated environment, which we modelled after the actual network conditions that we empirically measured in the test-bed depicted in *Figure 6.1*. For this purpose we reproduced the pictured topology and the measured throughput, delay, jitter, and packet loss between A and B, B and C, C and D in ns-3. For example, we measured a packet loss rate of around 2% on the link between B and C, and configured the same rate on the link we simulated. Please also refer to *Section 6.1.2*, where we evaluate more-or-less the same scenario within the actual physical test-bed.

Fitness Function Based on status reports sent by node B, C measured the achieved throughput, packet loss, delay, and jitter at the application layer, and calculated the transmission overhead between application-layer data and the actually transmitted data on the physical layer. The fitness measure we utilise for this scenario is derived from these measures. Please refer to *Section 5.3.5* for an introduction of the notation, the utilised functions, and reasoning that guided our fitness calculation.

The most important – and therefore most heavily weighed – component of the fitness function is the reception measure $\bar{R}$ which is inversely proportional to the packet loss, since we want to enforce correct transmission.

We calculate this value from S_A, the number of bytes sent at the application layer of the stack composition system running on node B, and A_A the number of bytes acknowledged by node C at the same position in its stack. S_A is directly measured at node B, whereas A_A is based on status reports sent by node B to C.

$$\bar{R} = \frac{A_A}{S_A}$$

Less important than correct transmission are the achieved throughput $\bar{T}$ and the transmission overhead $\bar{O}$, which we way at one quarter of the reception and calculate from the number of application-level bytes acknowledged by B per second T_A and the number of bytes sent at the application S_A versus physical layer S_P, respectively.

$$\bar{T} = 1 - 2^{-log_{10}\left(1+10^{-4}T_A\right)}$$

$$\bar{O} = \text{erf}\left(\frac{S_A}{S_P}\right).$$

The least important measure in the fitness function, which we therefore weigh at 15% of the reception, are the measures for delay $\bar{D}$ and jitter $\bar{J}$, which are calculated from the average of the measured raw delay D and jitter J values and the scaling values D_0, J_0 which we determined empirically before the experiment.

$$\bar{D} = \frac{D_0}{D}$$

$$\bar{J} = \frac{J_0 - J}{J_0}$$

From these measures and using the weighing function b introduced in Based on the considerations introduced in *Section 5.3.5*, we define the fitness F as

$$\mathsf{F} = \bar{R} \cdot b_{0.25}(\bar{T}) \cdot b_{0.25}(\bar{O}) \cdot b_{0.15}(\bar{D}) \cdot b_{0.15}(\bar{J}).$$

Configuration of the Evolution Logic We used two different algorithms for long-term evolution for this scenario. the Composition Tree Search and the Evolutionary Algorithm. We parametrised both algorithms to empirically determined values which we found to work best with respect to evolution quality and evolution speed. For the Composition Tree Search we uses a population size of 3, elite size of 1, search size of 1.7, modification probability 0.5, step δ 10, connector weight 5, and instance weight 100.

We configured the Evolutionary Algorithm with population size 3, mutation rate 0.9, mutation σ 2.0, and crossover rate 0.1. This parametrisation is rather different from those encountered in the literature, for reasons we further discuss below.

Experiment Normalization We performed 100 runs of the experiment in the simulator for each of the evolution logics, that is we repeated the same experiment 100 times for random seeds of $[0, 100)$. Every run lasted for 50 generations, and we performed only 2 sub-trials[ろ] as we expected the simulation to contain little noise.

Furthermore, we set the duration of the sub-trials to 30 s, i.e. identical to the traffic patterns period. One complete trial therefore takes 1 min to complete, and the two[は] new stack configurations that are trialled per generation can consequently be tested in 2 min.

Stack Modules For this experiment we provided the stack composition system running on node B with modules that implement the functionality of TCP, UDP, DCCP, IPv4. We modified the protocol specifications so that that arbitrary compositions of these modules are possible, e.g. TCP can be positioned on top of UDP, as illustrated in *Figure 6.2*. We further provided two dummy transport modules (**X**,**Y**), which add an undecodable dummy header to all data that passes through them. At the top of the stack we positioned the GCP, which is described in *Section B.2.2*, and which serves as a protocol multiplexer, i.e. binds to all of the aforementioned protocols and selects one of these for communication. The GCP provides one control, through which the evolution logic defines to which of the 6 possible targets all outgoing application packets are forwarded, i.e. which of its connectors to use. These target instances in turn process the incoming data packet and forward it to the next lower layer by means of a connector that can bind to one of 6 module instances. The evolution engine on node B can thus encode $6^6 = 46656$ different paths through the module graph, out of which only a fraction of $\frac{3}{5} \cdot \frac{1}{6} = 0.1$ is actually usable for communication, as also shown in *Figure 6.2*: We configured the stack composition system on node C such that it silently drops all incoming packets that do not conform to the normal Internet protocol stack and thus simulate a legacy client that does not support stack composition[に].

[ろ] See *Section 5.3.2* for an explanation of the stack composition system's experimentation methodology and a distinction between trials and sub-trials.

[は] The best found stack is kept over as elite.

[に] We still had to place an instance of the stack composition system at node C for practical reasons, as our measurements and thus fitness calculation depend on that node's status reports

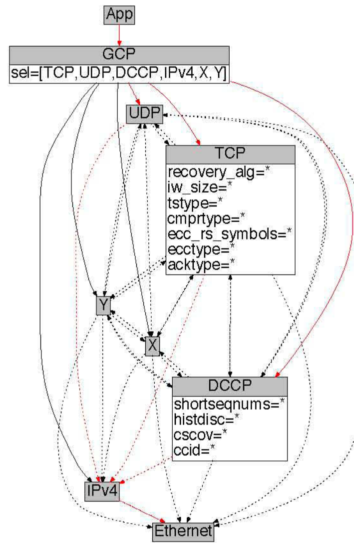


Figure 6.2: *The possible compositions of the stack modules, where red arrows indicate the valid paths through the stack. Black arrows denote those paths which result in communication failure when taken, either because the packets are dropped by the test modules, or the encoded packets are not decodable by the standard Internet stack implementations. Solid lines indicate fixed connections, dotted lines identify all possible connections for one connector. The text boxes denote the controls that can be configured by the evolution logic*

Most of the unusable stacks are not detectable by the stack composition system without actual experimentation. If the stack configuration encodes a loop, e.g. TCP is positioned on top of UDP, which in turn is connected to TCP again, the stack composition system detects this problem once the first outgoing packet is send, aborts the stack, and assigns it a minimal fitness value, as explained in *Section 5.3.2*. But for most other non-standard-conforming and therefore unusable stacks, such as TCP over DCCP over IPv4, the processing of outgoing data does not produce any errors[ほ], and the stack composition system on node B thus cannot notice any problems without actually performing on-line experimentation. The receiving node C silently

[ほ] All of these modules use the Unified Communication Interface, which enables arbitrary communication modules to communicate with each other, as described in *Section 3.7.3*.

drops packets encoded by these stack configurations, and the actually measured fitness for such stacks is zero.

Our implementations of TCP and DCCP both define several controls, which determine the protocol operation (see *Section B.2.1*). Due to these controls the size of the actual search space on which the evolution logic operates $7776 \cdot 128 = 995328$, but the ratio between usable and unusable stack configuration does not change.

Traffic Characteristics We deployed a test application on client A, which at intervals of 30 s sends 2 MB of bulk data at a constant rate of 300 kB/s via TCP to the stack composition system deployed on node B. The stack composition system running on node B transmits this data to node C using its current stack configuration as decided by its evolution logic. The corresponding stack is directly attached to a raw network socket, i.e. bypasses the operating system's IP stack. The stack composition system on node C again uses TCP to forward the received data to Node D.

Since we wish to focus on the abilities of the evolution engine to adapt the stack, we synchronised the operations of node A and B, i.e. we guarantee that node A performs exactly one send operation and during every sub-trial performed by node B^.

Results

The adaptation behaviour encountered in this experiment is depicted in *Figure 6.3* for the Composition Tree Search and *Figure 6.4* for the Evolutionary Algorithm. For sub-figure (a) we grouped the runs per generation according to the fitness the best stack found achieved in that run, and calculated the average over each of these 10 sets. The average fitness of 10% group that achieved the highest fitness is denoted by the solid green line, followed by the average fitness of the next-best group, and so on to the worst-performing 10% shown by the red line. Sub-figure (b) was generated by calculating the average of 10% of runs that exhibited the highest fitness (again given in green), then the average of the best 20%, and so on, up to the average over all except the worst 5% of runs (denoted by the red line). The dotted green and red lines in both sub-figures show the fitness of the overall best and worst stack found during the experiment. Sub-figure (c) shows the fitness distribution of the tested stacks during the initial randomly created generation

^ This step allowed us to reduce the overall time needed to perform the simulations, as without the stack composition system would discard the results and re-schedule all sub-trial during which not enough data was transferred.

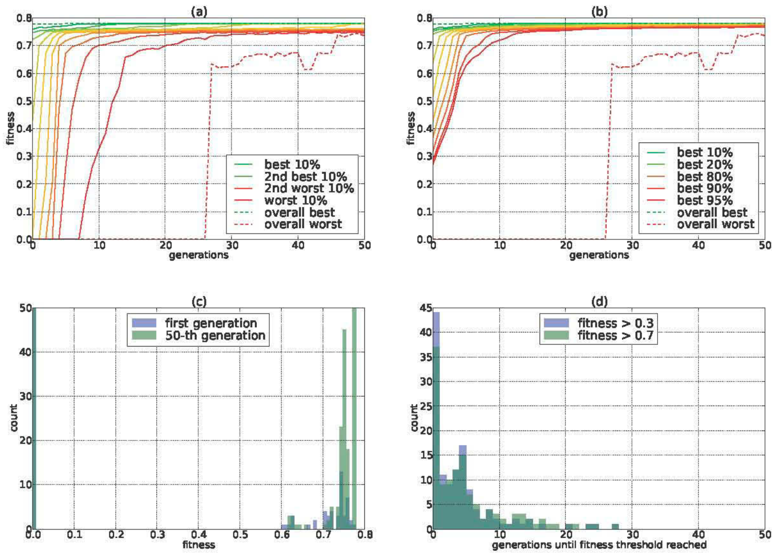


Figure 6.3: *Adaptation behaviour of the Composition Tree Search for the Internet Stack composition scenario.*

(blue), and during the last generation (green). Sub-figure (d) shows the histogram of the time needed to reach a fitness threshold of 0.3 and 0.7 in blue and green, respectively. The time is given in generations, where one generation correlates to approx. 2 min of real time.

As illustrated in sub-figure (d), both algorithms managed to generate an usable stack in less than 10 generations, i.e. about 20 min, during 95% of all runs. Even in the worst case encountered during our 100 runs, both methods achieved a positive fitness after at most 27 generations, which corresponds to around 50 min. Since (as described above) only 10% of all possible compositions can achieve a positive fitness, the probability x for reaching a positive fitness in 95% of all trials is $0.9^x = 0.05 \iff x \log 0.9 = \log 0.05 \implies x \approx 28.43$, both algorithms perform at least as good as random probing, with high probability even better. Both algorithms thus perform better than the average specialised search algorithm (see *Section 2.6.5*). The fitness of the candidate stack configurations developed by these two algorithms also improved over time, which was not the case for random probing. As shown in sub-figure (c), the stacks tested in the final generation achieved a significantly higher fitness than the initial generation. Since the initial generation

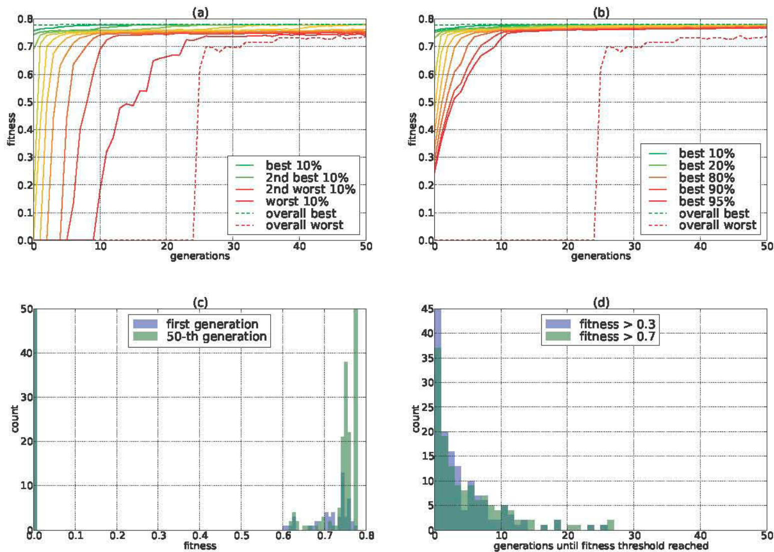


Figure 6.4: *Adaptation behaviour of the Genetic Algorithm for the Internet Stack composition scenario.*

is randomly generated, its histogram also gives a good indication of the fitness distribution of the entire search space.

Summary

In this scenario we made the stack composition system to produce a feasible network stack out of several, almost arbitrarily connectible modules. We demonstrated that the stack composition system is able to find a near-optimal stack configuration within a limited amount of time, and an usable stack within an even shorter period, at a high confidence level. The time needed for adaptation is however far from instantaneous and only adequate for scenarios which do not change quickly and where sufficient time for experimentation and adaptation is available. The need for an additional layer of adaptation, for example the mid-term population selection (see *Section 4.2*) or the short-term in-stack redirectors (see *Section 4.3.1*), thus became apparent.

Additionally the importance of normalization for our fitness calculation became apparent. The two sub-trials we performed were insufficient to ef-

fectively normalize the fitness measurements, resulting in a noticeable fluctuation caused by e.g. variations in the error rate. And even though we performed five sub-trials in the real-world test bed, the influence of noise was even higher. Consequently the system was unable to find the true optimum, as the impact some settings, such as whether or not to use TCP timestamps, had on the fitness was lower than the variation caused by environmental effects. We therefore conjecture that the measurement accuracy, which directly affects the precision of the fitness calculation, also limits the evolution quality. The higher the error in the fitness assessment, the lower the ability of the system to reliably differentiate between similar – but not identical – utility.

The optimal parametrisation we found for the Evolutionary Algorithm through trial-and-error experimentation is rather different from the values described in textbooks or reported as optimal for specific applications.[140,177] We assume that the reason for this is threefold. First of all, our implementation differs from the textbook implementations – please see the description the our algorithm in *Section 4.1.3* – and thus the optimal parametrisation also differs. Secondly, the cost of experimentation in our case is rather high, thus the detrimental effect caused by a large population size, i.e. the long time needed per generation to experiment with every stack configuration for that generation, seems to outweigh the reduced evolution quality. Lastly, the search space for this scenario is rather small. For more complex scenarios, i.e. a higher variety of modules, the optimal parametrisation will likely differ.

6.1.2 Scenario 1B: Composition of the Internet Protocol Stack – Physical Test-Bed

In this scenario we empirically validate that the simulation results found for scenario 1A resemble reality, i.e. that the results reported there accurately replicate the adaptation behaviour when the system is deployed in the physical test-bed. As the set-up closely resembles the previous one, we only discuss the differences to scenario 1A and the results in here. Due to the high work load for setting up, executing and monitoring the experiment, we only performed 10 experiment runs for every evolution logic we tested. We further configured the stack composition system to average the fitness over 10 sub-trials to compensate for higher measurement inaccuracies[ɛ].

[ɛ] Whereas we did try to reproduce the actual physical environment as closely as possible in the simulator, the actual packet loss rate, etc.in reality were rather unpredictable, and did not even conform to those we encountered a few days earlier when we prepared the simulation experiments.

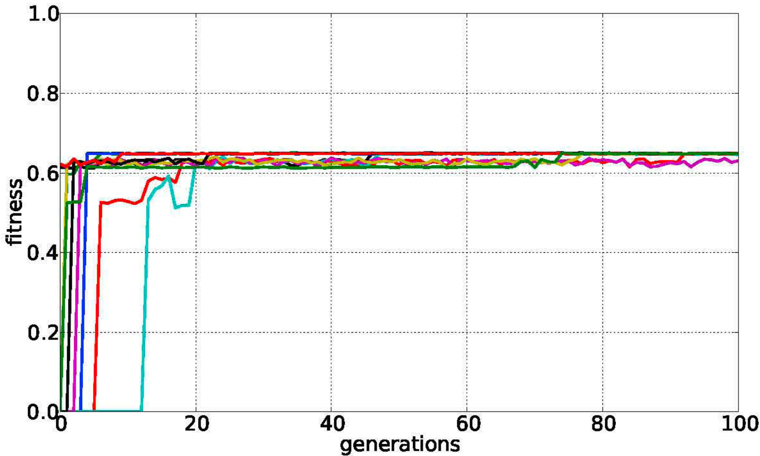


Figure 6.5: *The fitness achieved by the ten runs performed in the real physical test bed.*

The fitness for the best stacks found by the Composition Tree Search during these experiments is depicted in *Figure 6.5*. While the number of trials is too low to make any definitive statements the higher level of noise in these experiments is noticeable, even though we averaged over ten sub-trials instead of two as in the simulation. The fitness values themselves are noticeably different from the ones measured during the simulation, due to the difference in traffic and network conditions. In particular the achieved throughput, delay and jitter on the real-world test bed not only differed from the simulation results, but also fluctuated significantly.

During the limited number of experiment experiment runs we performed, the system was able to adapt the network stack to the specified requirements in the same way as during the simulation. There was no indication that the results from the simulation are not applicable to reality, but further experimentation is needed to achieve certainty.

We further measured the impact that a deployment of the stack composition system has on system performance. Since our current implementation is single-threaded, non-operational tasks, such as the application of the evolution logic or the re-composition of the stack induce overhead. Our measurements on actual router-class hardware do however indicate, that this

impact is negligible, as these actions are infrequent, and stack operations are prioritised. The longest delay we measured in our experiments – apart from disk activity due to logging, which is disabled in a deployed system – was caused by the Evolutionary Algorithm and in the range of tens of milliseconds. As the evolution engine is independent of the actual stack operations, the necessary processing could further be handled in a separate thread on multi-core processors.

6.1.3 Scenario 2: Error Correction & Compression

After establishing the capability of our system to compose network stacks out of protocol modules and driven by a utility function in the previous scenario, we now turn our attention to the problem of finding a module's optimal parameter configuration and placement within the stack. Specifically we present a scenario in which the stack composition system has to decide on the optimal configuration of a module which provides error correction and compression, and where to place it inside an otherwise fixed stack composition. In this scenario the utility of the stack depends on environmental conditions which are unknown at design time. We again let the system start from a randomly initialized state, and measure if and how fast the stack performance approaches the optimum.

Experimentation environment

In this scenario we again utilise a wireless set up, as radio communication is often plagued by packet corruption and loss, the severity of which depends on the current signal-to-noise ratio, interference level, and traffic conditions. Since these influences vary widely even within the same network, the decision of what countermeasures to take is best taken at runtime, and could thus provide an almost ideal test case for our stack composition system.

Here forward error correction plays a vital role as it is the most widely deployed countermeasure against problems caused by noise in wireless communication channels. The introduction of error-correcting codes can reduce some types of data corruption and resulting loss, but incurs additional overhead by adding redundancy, i.e. overhead, to the transmitted data. The resulting trade-off between achieving a higher probability of successful transmission and a reduced bandwidth caused by adding more error correction symbols causes the optimal configuration to depend on the current network conditions and application requirements. As these conditions are unstable, optimal performance requires continuous adaptation of the employed error correction method based on e.g. the measured reception quality. Conse-

quently network cards compliant to modern WLAN standards such as IEEE 802.11n[165] employ heuristics to chose from one of multiple error-correcting codes variants based on the measured network conditions.

For this scenario we performed experiments only in the simulation environment, introduced in *Section B.1.2*, as this was the only way to get us full access to the link layer and provided us with functionality to introduce errors at a configurable rate: The wireless interface cards available to us did not allow us to modify the employed error-correcting codess, thus we were unable to realise this scenario on the physical test-bed.

Network Topology & Traffic Characteristics For this scenario we again used the simulation environment described for the Scenario 1A and depicted in *Figure 6.1*. Node A was again configured to send 2 MB of bulk data at a constant bit rate of 300 kB/s to node D, and repeat this process at an interval of 30 s. The transmitted data consisted of concatenated, but uncompressed HTML pages. The maximum packet size was limited by the MTU of the simulated VDSL link, i.e. 1350 bytes. The rate error module of ns-3 enabled us to introduce random uniformly-distributed byte errors into the packet data. NS-3 on its own only reports that a packet is corrupted, thus we extended it by randomly corrupting bits in the packet once it had traversed the IPv4 module, in accordance to the error probability. For this scenario we set the byte error rate (BER) to a relatively high value of 0.001, so that communication is only feasible if an effective error correction method is employed.

Fitness Function Apart from the sensor measures introduced for Scenario 1A, we use the computational overhead $\bar{C}$ caused by adding error correction and compression, i.e. the time needed to perform these operations in relation to the time needed for the normal stack operations. We further re-define the overhead measure as $\bar{O}c_0\frac{S_A}{S_P}$, and emphasize the effect of overhead compared to the error-function-based measure we used for the previous scenarios. The constant c_o was derived empirically to guarantee that the possible values of $\bar{O}$ fall in the $[0, 1]$ range and depends e.g. on the application payload. We further defined the fitness function as

$$\mathsf{F} = \bar{R} \cdot \bar{C} \cdot \bar{O} \cdot b_{0.25}(\bar{D}) \cdot b_{0.2}(\bar{T}).$$

This function, on which the results discussed below are based, puts a strong emphasis on correct transmission while incurring little transmission overhead. To test that the system is also able find the correct settings for other weights, we repeated the experiments below for function

$$\mathsf{F} = \bar{C} \cdot \bar{O} \cdot b_{0.25}(\bar{D}) \cdot b_{0.2}(\bar{T}) \cdot b_{0.1}\bar{R},$$

In this case the adaptation dynamics were mostly identical, thus we omitted a thorough discussion below and only describe how the optimal stack configuration differs.

Configuration of the Evolution Logic We performed our experiments using the Evolutionary Algorithm and two differently-configured versions of the Composition Tree Search algorithm. For Evolutionary Algorithm we used a population size of 3, elite size of 1, mutation probability of 0.9, mutation σ of 2.0, and a crossover probability of 0.1. We configured the 1st variant of the Composition Tree Search with a population size of 3, elite size of 1, a search size of 1.7, the modification probability of 0.5, step δ of 10, connector weight 5 and instance weight 100. The 2nd variant used a connector weight of 2, but was otherwise identically configured. Additionally we applied the random probing algorithm as baseline for comparing the adaptation behaviour.

Experiment Normalization We configured the system to perform 10 sub-trials for every stack tested, and performed at 50 simulation runs per evolution logic in ns-3. We set the duration of a sub-trial to 30 s, i.e. identical to the data transmission interval, so that one complete trial takes a total of 5 min to complete. To test the two new stack blueprints created per generation the system therefore needed approx. 10 min. We stopped the runs once 100 generations had been tested.

Stack Modules The network stack for this experiment was composed of protocol modules for IPv4, Ethernet, TCP, UDP, and DCCP, for which we enforced layering according to the IETF standards by means of the module specification language described in *Section 3.4.1*. We further introduced the Codec module described in *Section B.2.2*, which the stack composition system was able to place either on top of the transport protocols, below them, or next to them on the transport layer, thus bypassing the transport protocols. We again employed the GCP module to select the underlay for all outgoing communications, which encoded ten different paths through the stack.

The Codec module provides compression and error correction, which the stack composition system was able to configure through the exposed control values. It offers four controls, two to select the employed error-correcting codes and compression method, and two for parametrising them. As explained in *Section B.2.2* and *Section B.2.2*, the module provides three compression methods, uncompressed, RLE, and DEFLATE, and three error correction methods, CRC32, Hamming codes, and Reed-Solomon codes.

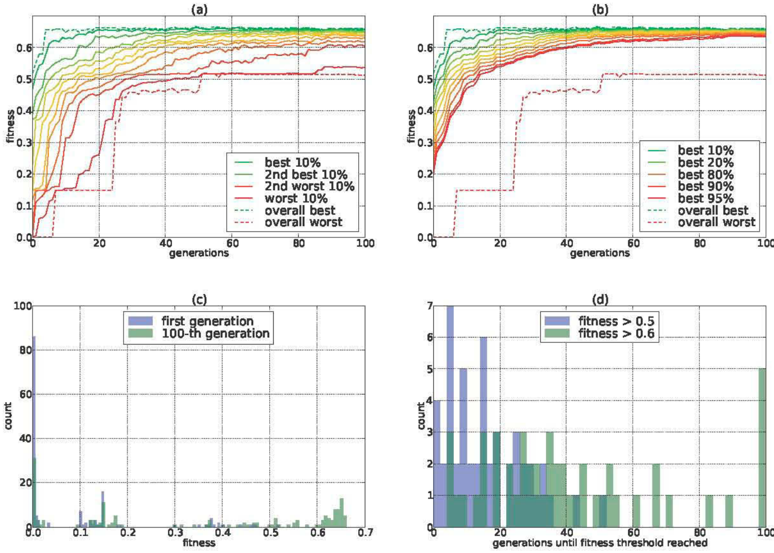


Figure 6.6: *Adaptation behaviour of the 1st configuration of the Composition Tree Search for the error correction and compression scenario.*

Since the controls are independently modifiable, the parametrisation controls can embody different meanings such as the compression level or be totally meaningless, e.g. in the case when CRC32 is selected as error detection method. For Reed-Solomon, the parameter represents the number of redundant bits per 64-bit block, out of $\{2i \mid 1 \leq i \leq 14\}$. For deflate compression, the parameter encodes the compression level in the range $[1, 9]$

Overall the configuration space encompasses a total of $2^3 \times 3 \times 9 \times 3 \times 14 = 90720$ different parametrisations, all of which are actually usable for communications. But since the error rate in the network is rather high, the data payload is well compressible, and the fitness function favour little loss and low overhead (as described above), only very few of the configurations can attain a high fitness value.

Based on the described scenario we expect that the stack composition system will configure the system to use Reed-Solomon codes, configured to guarantee near-perfect transmission with a low number of redundant symbols, and use the deflate algorithm for compression, configured to one of the highest compression levels.

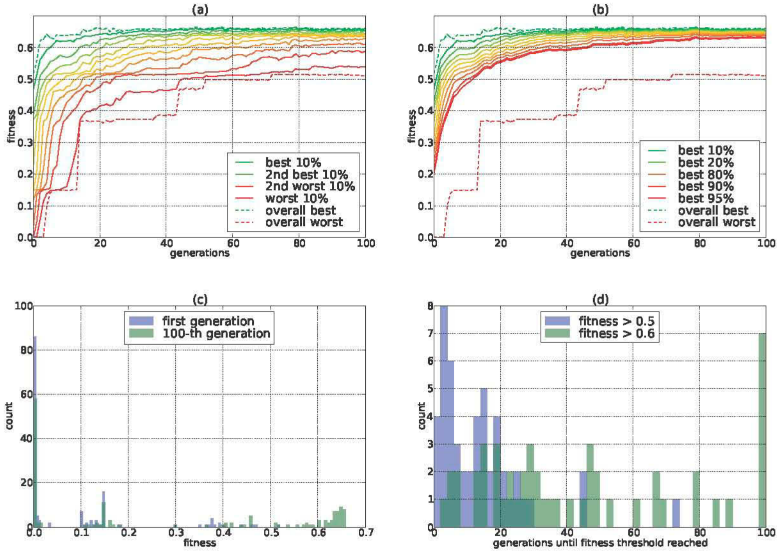


Figure 6.7: *Adaptation behaviour of the 2nd configuration of the Composition Tree Search for the error correction and compression scenario.*

Results

For this scenario we again present separate graphs for all four evolution logic variants we tested in *Figures 6.6, 6.7, 6.8 and 6.9*. These graphs detail the adaptation behaviour in the same way as described for the preceding scenario on 152. As opposed to the previous scenario, however, this behaviour varied strongly between the tested algorithms. Both variants of the Composition Tree Search achieved a fitness greater than 0.5 within at most 25 generations, as shown in sub-figure (d) of *Figures 6.6 and 6.7*. The Evolutionary Algorithm, however, failed to reach this threshold after 100 generations in 7 out of the 50 runs we performed, as shown in *Figure 6.8*. For the higher fitness threshold of 0.6, the differences were even more pronounced. The 1st variant of the Composition Tree Search missed the target in 5, the 2nd variant in 7, and the Evolutionary Algorithm in 34 out of 50 runs. This is even worse than for the random probing method, which only failed in 21 runs. These differences in behaviour are likely due to the differences between the algorithms – and different parametrisations therefore – in treating continuous controls, i.e. whether they are likely to make larger or smaller changes to

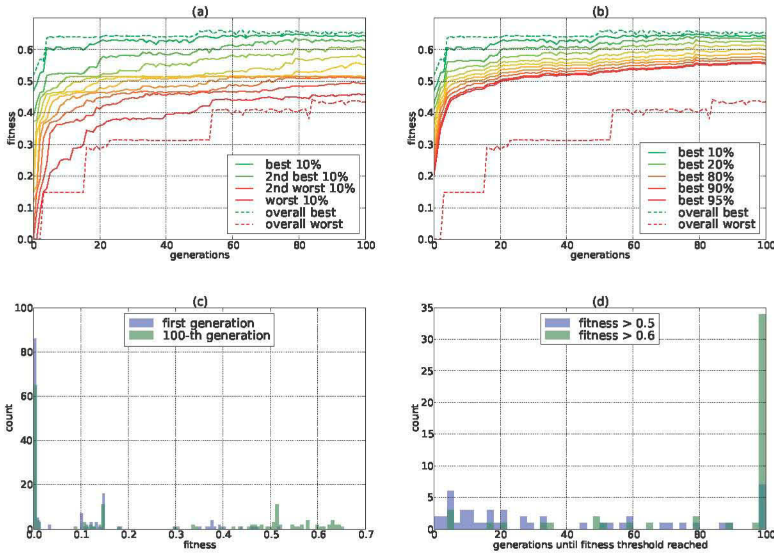


Figure 6.8: *Adaptation behaviour of the Genetic Algorithm for the error correction and compression scenario.*

such control values, etc.

Sub-figures (c) illustrate the importance of error correction in this scenario. A majority of the 150 randomly-created stacks in the initial generation exhibited an extremely poor fitness of less than 0.1. A random inspection of the stack configurations in this range and their sensor records showed that they employed either insufficient or no error correction at all, or placed the Codec module too high in the stack, which caused the checksum test in the underlying modules to fail and the packet to be dropped. A low, but non-zero fitness requires that at least some packets passed correctly, indicating an insufficient forward error correction method setting, or in very rare cases, several uncorrupted packets. Since the GCP module employs its own checksum algorithms, failure to detect corruption was highly unlikely[5].

Sub-figures (a) clearly visualize the effects a one-bit change in the stack configuration can cause. When a nominal control changed, e.g. a better er-

[5] We did only explicitly check whether the received data matches the transmitted data during one run, during which we detected no such corruption. This does however not prove the total lack of undetected but corrupted packets during all trials, since the received data was silently dropped for performance reasons

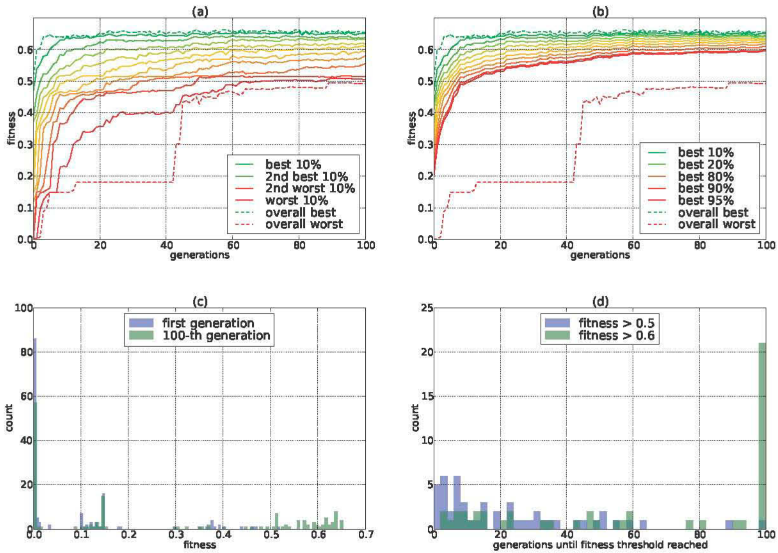


Figure 6.9: *Adaptation behaviour of of the random probing algorithm for the error correction and compression scenario.*

ror correction method was chosen, or a different call path through the stack selected by the GCP module, the fitness changed sufficiently to introduce a step-like effect in the graphs. Since we performed only 50 runs per evolution logic variant, each line in this graph represents the average over 5 runs, thus these artefacts are not hidden by normalization. Sub-figures (a) also show that the 1st variant of the Composition Tree Search on average exhibits precision of adaptation than the 2nd variant, i.e. it approaches the optimum more closely in the long run. The 2nd variant however shows a quicker improvement earlier in the experiment, and e.g. passed the fitness threshold of 0.3 in at most 15 generations, while the 1st variant took generations to achieve the same feat. Surprisingly random probing was the second-fastest of all methods we tested in the initial phase of adaptation, as it needed only slightly longer than the 2nd variant of the Composition Tree Search generations to reach this threshold in 90% of all runs. After this point, all Composition Tree Search variants surpassed the random probing method. The Evolutionary Algorithm disappointingly failed to beat the random search in any category. This might be due to an inefficient configuration of the algorithm, but our (limited) fine-tuning efforts failed to arrive at better-performing variant.

At the end of the experiment, we analysed, which stack configurations were most prevalent amongst the best stacks found during all runs. For the random probing algorithm, we noticed that out of 50 stacks, the best 15 all directly used the codec module, bypassing all transport protocols. Only 4 out of the top 25, i.e. 50%, of stacks used UDP. The top 40 out of 50 stacks employed Reed-Solomon error correcting codes, the next best 7 stacks only used CRC32, and the worst-performing two again Reed-Solomon. In all of these cases, the module was positioned directly on top of IPv4. The top 30% of all stacks used either two or four redundant symbols, as adequate for the rather low error rate. The top four stacks used two symbols, the next three stacks used four symbols per 64-bit block.

While all stacks in the top 50% utilised the deflate algorithm for compression, the compression level was unexpectedly distributed: 4 of 10, i.e. the top 20% of stacks used the highest level 9, the 3rd-best used level 7, the 5th- and 7th-best stacks used level 8, the 6th- and 10th-best used 5, and the 9th-best used 4. The configurations of the 2nd- and 3rd-best stack, while otherwise totally identical, differed only in the compression level, where level 9 resulted in a fitness of 0.652831, and level 7 in fitness 0.652011, i.e. a rather low fitness difference. Incidentally, the 4th-best stack was identically configured as the 3rd-best, but only achieved a fitness of 0.638713. Measurement noise thus had a higher influence than the compression level.

The same trends are also reflected in the results for the Composition Tree Search variants, but are even more apparent. Here the compression level was even more evenly distributed between levels 5 and 9. The best performing 21 out of 50 stacks for the 1st variant of the Composition Tree Search used two Reed-Solomon symbols for every block. For the 2nd variant, the top 12 stacks also used two symbols, followed by 9 stacks using four symbols. Likewise, all of the top 50% of the found stacks bypassed the transport protocols and send data directly via the codec module positioned on top of IPv4. The Evolutionary Algorithm included a higher variance of stack configuration, which corresponds to the lower overall fitness of achieved by these stacks.

All stacks that actually used error correction placed the module directly over IPv4, in accordance with our expectations. Since we introduced bit errors in the simulator after the IPv4 frame was decoded, and since the higher-layer protocols all drop corrupted packets due to the resulting checksum mismatch, any other position would have been inefficient, i.e. would add overhead but offer no benefit.

Summary

By means of this scenario we showed that the stack composition system is able to configure the modules in the stack such as to maximise the utility as measured by the fitness function. The system was able to produce stack configurations which apply close-to-optimally configured error correction and compression functionality and place it at the correct position within the stack. Our results highlight the importance of not only choosing an appropriate algorithm, but also configuring it properly for optimal results. In particular the different handling of continuous controls, i.e. the probability distribution of the new value in relation to the previous value of such a control, affects the performance of the algorithms. A propensity for large changes, i.e. a high delta between the old and new value, improves the initial adaptation speed, but reduces the precision or fine-tuning of the value once a close-to-optimal solution is found. An adaptation of the evolution logic's parametrisations based on the achieved fitness as we propose in *Section 4.1.4* therefore seems to warrant further research. Overall, even the less effective methods were still able to significantly improve the fitness over the duration of the experiment. We further notice the effect of measurement noise on the evolution quality. The system was unable to find the optimal compression level, as the fitness difference was lower than the measurement noise. Overall, higher compression levels were more likely to be chosen than lower ones, but the variance was rather high.

6.1.4 Scenario 3: Bias Towards Specific Configurations

After analysing the previous two scenarios we became interested in exploring the adaptation behaviour under conditions were the difference in fitness between different configurations are more pronounced. Since the number of actual protocol modules available to use is rather limited, we decided to use a slightly synthetic scenario for this purpose, in which we artificially "enhanced" the performance difference between e.g. DCCP and UDP, and thus provide a richer play-field for our algorithms, which we could otherwise only achieve through disproportionate implementation and set-up efforts.

Experimentation environment

This scenario is a variant of the previous one and as such we performed it in the same environment. As we only describe the differences to Scenario 2 in here, please refer to 6.1.3 further details.

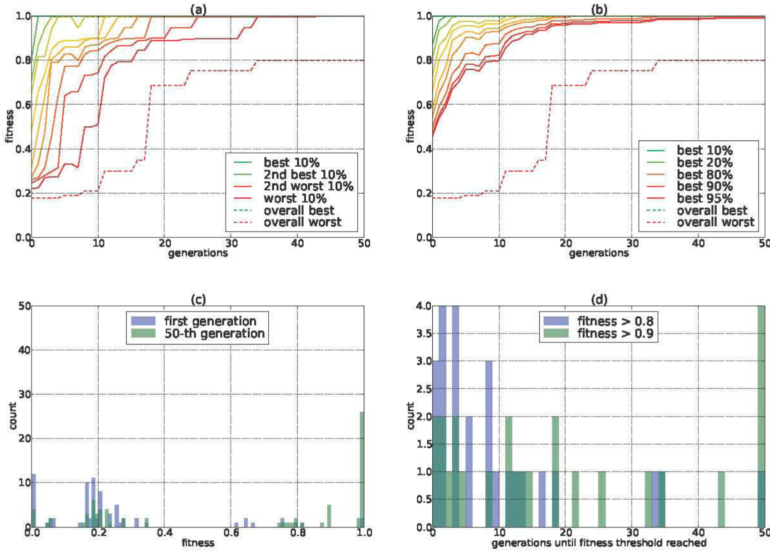


Figure 6.10: *Adaptation behaviour of the 1st configuration of the Composition Tree Search for the biased scenario.*

Fitness Function The major difference to the previous scenario is the definition of the fitness function. We intended to increase the difference in fitness for the individual transport protocols, as during the previous experiments in particular the decision between DCCP and UDP was ambiguous. The fitness difference between otherwise identical stacks that used either DCCP or UDP was marginal. For this experiment we intentionally introduced a bias between the protocols[ŋ]. The fitness function was defined as follows:

$$F = \bar{R} \cdot \bar{B} \cdot b_{0.5}(\bar{O}),$$

which again uses the measures for reception quality $\bar{R}$ and overhead $\bar{O}$ based the error function as introduced for Scenario 1A. We further defined a protocol bias B as shown in the following table.

Protocol	UDP	DCCP	TCP	IPv4	otherwise
B	1	0.9	0.8	0.2	0

[ŋ] In reality such a bias might actually occur e.g. due to policy constraints

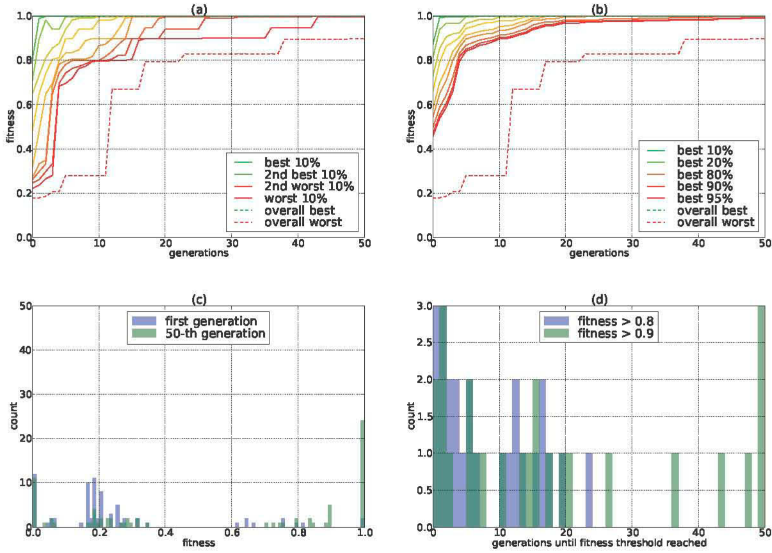


Figure 6.11: *Adaptation behaviour of the 2nd configuration of the Composition Tree Search for the biased scenario.*

Configuration of the Evolution Logic After noticing the importance of the correct parametrisation of the evolution logic for the adaptation behaviour in Scenario 2, be decided the investigate this matter further by looking at several different parametrisations of the same algorithm. We further employ a dynamically adapting variant of the Composition Tree Search, which changes its parametrisation based on the fitness of the best stack found so far. We hope that a method which covers a larger area of the configuration space at the start and then focuses on fine-tuning the found solution later in the process can improve the overall evolution quality and evolution speed. In addition to the methods and configurations introduced in the previous scenario, we therefore provide the following parametrisations of the Composition Tree Search algorithm, which exhibited the best adaptation behaviour in the previous experiments and thus warrants more detailed exploration. For Evolutionary Algorithm we again used a population size of 3, elite size of 1, mutation probability of 0.9, mutation σ of 2.0, and a crossover probability of 0.1. We configured the 1st variant of the Composition Tree Search with a population size of 3, elite size of 1, a search size of 1.7, the modification probability of 0.5, step δ of 10,

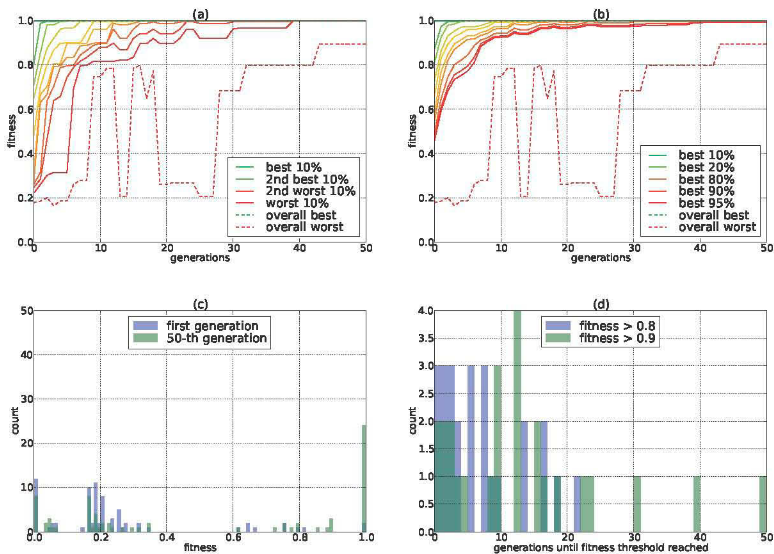


Figure 6.12: *Adaptation behaviour of the 3rd configuration of the Composition Tree Search for the biased scenario.*

connector weight 5 and instance weight 100, as before. The 2nd variant used a connector weight of 2, but was otherwise identically configured, also identically to the previous settings. We introduced a 3rd variant, for which the search size was defined as 1.2, but which was otherwise identically configured as the 2nd variant. Additionally we employed an adaptive variant, which switched between two different configurations depending on the fitness of the best stack found so far. If this fitness value was below 0.65, it used the configuration of the 2nd variant, and otherwise the configuration of the 1st variant. Finally we also experimented with a random search algorithm, which generated two new stack configurations totally at random every generation and kept the overall best one around. This algorithm serves as a performance-baseline for comparison.

Experiment Normalization Since we noticed in the previous scenario that the algorithms were unable to the find the best compression level due to a high noise content in the measurements, we increased the number of sub-trials to 5 sub-trials for every stack tested, and performed at 25 simulation runs per evolution logic in ns-3. We set the duration of a sub-trial to 30 s, i.e.

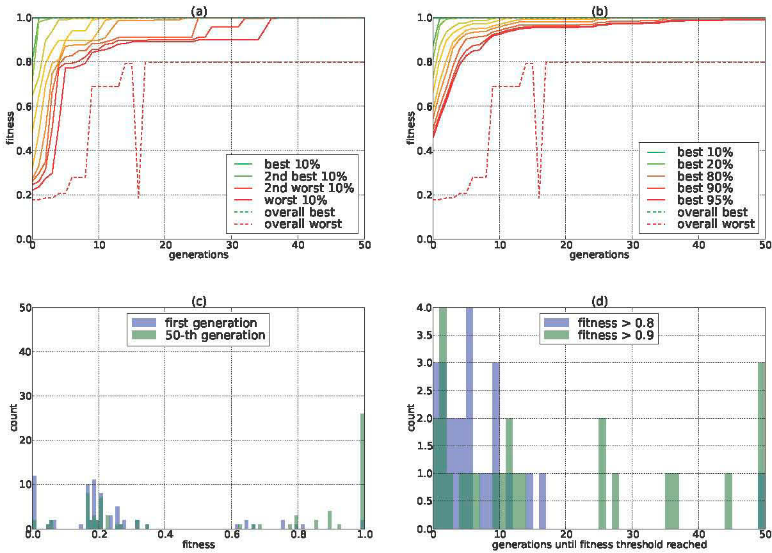


Figure 6.13: *Adaptation behaviour of the adaptive variant of the Composition Tree Search for the biased scenario.*

identical to the data transmission interval, so that one complete trial takes a total of 5 min to complete. To test the two new stack blueprints created per generation the system therefore needed approx. 5 min. We stopped the runs once 50 generations had been tested.

Results

We again present separate graphs for all configurations of the tested evolution logics. *Figures 6.10, 6.11 and 6.12* illustrate the behaviour of the three variants of the Composition Tree Search we explored. *Figure 6.13* details the behaviour of the adaptive variant of the same algorithm, and *Figure 6.14* of the Evolutionary Algorithm. The results for the random probing algorithm are shown in *Figure 6.15.*

Sub-figures (a) again clearly illustrate the impact the selection of different algorithm for evolution or a different configuration of this algorithm can have. Even our relative slight changes to only one parameter resulted in sufficiently different adaptation behaviour to be clearly noticeable in the graphs. The 2nd parametrisation of the Composition Tree Search, whose behaviour

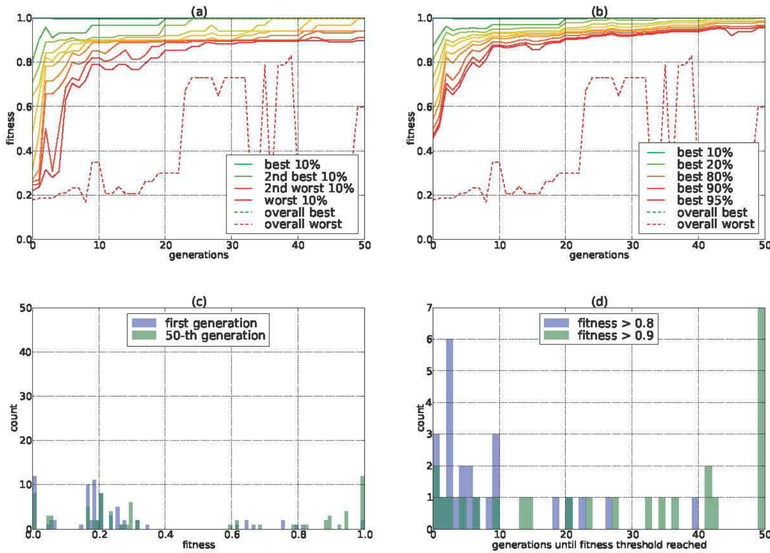


Figure 6.14: *Adaptation behaviour of the Evolutionary Algorithm for the biased scenario.*

is shown in *Figure 6.11*, was on average quicker to reach a fitness level of up to 0.7, whereas the 1st variant performed slightly better in the long run. The worst runs of the 3rd variant remained at a fitness level of around 0.9 for a relatively long time, but managed to reach the optimal configuration in one step, which is likely due to the higher probability of randomization.

More importantly, the combination of the 1st and 2nd variant, i.e. the adaptive variant shown in *Figure 6.13*, was able to show a slight improvement in the overall adaptation performance compared to the 2nd variant on its own. It however also also failed to achieve the fitness threshold of 0.8 in one case, just as the 1st variant, as shown in sub-figures (d). The adaptive re-configuration of the algorithm failed to fully achieve our expectations, as the result did not combined the advantages of both parametrisations – high adaptation speed at the beginning of the process versus good fine-tuning behaviour: It exhibited a positive impact on the adaptation behaviour, i.e. the behaviour above and below the threshold strongly resembled the chosen variant and thus further investigation of this approach is warranted. The overall probability of reaching the threshold of 0.8 did however not improve, as the adaptive variant also failed to reach this threshold in one case after 50

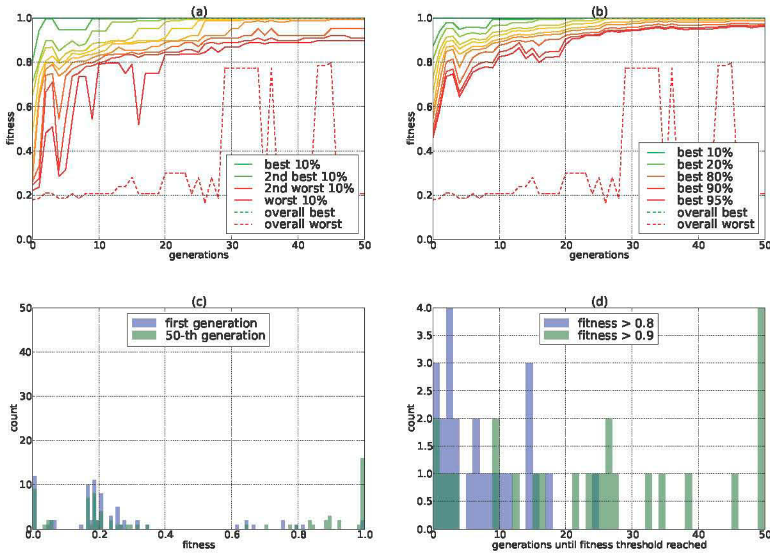


Figure 6.15: *Adaptation behaviour of the random probing algorithm for the biased scenario.*

generations, while the 2nd variant consistently surpassed it in at most 25 generations. Whereas this effect might might be attributable to the low number of performed trials, i.e. an outlier case, our current results in this aspect are inconclusive.

In 95% of all runs, the Composition Tree Search managed to find the optimal configuration in less than 40 generations, for all variants except the 2nd which needed 43. The Evolutionary Algorithm achieved this feat in only around 60% of all runs.

Overall, only the 3rd variant managed to reliably surpass the fitness threshold of 0.9, which it only failed to reach in one of the 25 runs within the 50 generations. The other variants failed to pass the threshold in the three (2nd and adaptive variant) or four (1st variant) cases, the Evolutionary Algorithm in 7 out of the 25 runs.

When comparing the results of the specialized methods with the randomized search, the variants of the Composition Tree Search show a significant advantage both in adaptation precision and speed. It is a sad fact, that the Evolutionary Algorithm performed worse than the randomized search for this particular problem. It should be noted that that since the random

probing algorithm also includes an elite of size one, the histogram in subfigure (c) is also biased towards the higher-performance stack configurations. Contrary to the other algorithms this bias is less clear.

Summary

In this scenario we evaluated whether the stack composition system is able to arrive at the true optimum in a limited amount of time. When the Composition Tree Search algorithm was used, the probability of success was sufficiently high, i.e. only one out of 25 runs failed to reach the optimum within the allotted time of 50 generations, approx. 250 min. This rather long period of more than two hours seems unrealistic for many applications where the environmental and traffic conditions are unstable. This problem may hopefully be alleviated by the introduction of short- and mid-term adaptation methods if similar conditions.

Additionally we showed that dynamic run-time re-configuration of the evolution logic's parametrisation can positively influence the adaptation behaviour, as we expected. After all, modifying the trade-off between exploration and exploitation during operation is a well-established technique employed by many search algorithms, e.g. simulated annealing. The effect was however less pronounced than we would have had hoped.

6.1.5 Summary and Conclusions regarding Long-Term Stack Evolution

After evaluating the adaptation behaviour of the evolution engine and the algorithms we developed for it in more-or-less realistic scenarios, both in simulation and to some extend on a physical test bed, we can conclude that our approach is feasible for at least some[a] application areas, when our preconditions are met, i.e. the network and traffic conditions that influence the performance stay stable enough for a sufficiently long period of time.

Taken together, our experiences with the different scenarios show that adaptation does occur, i.e. that the system is capable of improving the stack configuration over time and autonomously progress towards the optimum. Probably due to the non-deterministic nature of the algorithms we implemented, the true optimum was not reliable reached, but a close approximation of the optimum was attained with a high likelihood within the time we allotted for each scenario. For the tested scenarios the Composition Tree

[a] For generalization the number and variety of the different scenarios we explored is far too low.

Search algorithm appeared to be best suited, but these results are not generalisable to other scenarios. From The characteristics and in particular the variation of the behaviour of the algorithms we tested we gather that both the precision and speed of evolution is largely determined by the scenario to which the system is applied, i.e. depends on the specifics of the configuration space and the fitness landscape.

The provision of multiple algorithms to choose from and the ability to configure them appropriately is therefore sensible: The parameters offered by algorithms such as the Composition Tree Search bias the search either towards random exploration or local search in vicinity of known good solutions, define the size of the local search area, etc. Depending on the properties of the configuration space and the fitness landscape, different parameter sets were advantageous, but the exact criteria of how to choose these criteria are still unknown. In particular, experiences in other areas for this purpose are not easily applied to our problem, as exemplified by the low population size we are forced to use for the Evolutionary Algorithm shows (see discussion for Scenario 1A). It is also unclear under what conditions which algorithm behaves best. Our rudimentary experience with dynamic reconfiguration of the Composition Tree Search suggests that the dynamic run-time re-parametrisation of the evolution logic might be advantageous. Further research regarding potentially more appropriate algorithms, how to configure them, and how to dynamically adapt them at run-time seems worthwhile and necessary.

When translated to real time, the evolution speed varied widely and depended on the scenario. Here the difficulty of reliably measuring a stack's fitness is just as important as the composition and difficulty of the search space. The length of one sub-trial needs to be long enough to encompass and reliably capture the situational and performance characteristics needed for fitness derivation. And since fluctuations in the measurements cannot be completely avoided, often multiple sub-trials need to be performed, the number and duration of which entirely depends on the scenario.

Nevertheless, in our experiments the optimum was often reached within minutes, but to guarantee a close approximation of the optimum with a probability of 0.9 or 0.95, between 40 min were needed for Scenario 1A and 100 generations times 10 min per trial for Scenario 2, i.e. approx. 16 h to reach a close to optimal fitness with 90% probability. These results clearly show that our approach is not appropriate whenever quick adaptation is needed, but this was obvious from the beginning. As we intend this approach to be applied when sufficient time for testing and experimenting is available, when the situation is sufficiently stable, and in combination with other mechanisms for short- and medium-term evolution, we do not consider this a grave

limitation. The need for such additional short- and mid-term adaptation layers is apparent for scenarios were the conditions cannot be expected to remain stable. We assume that our own approach for mid-term adaptation – which we discuss in the following section – allows for the long-term adaptation process to be interrupted when the situation of the network changes sufficiently, and to be restarted again later once sufficiently similar conditions recur, which should help to further increase the range of applicable scenarios for our long-term evolution approach.

6.2 Evaluation of the Situational Classification Approach

After focusing the performance of the long-term evolution logic in the previous experiments, we now turn to the mid-term adaptation logic, the situational classifier. We explore the effects of switching between four populations based on a fixed and exact decision criterion using the matrix-based population selection method we introduced in *Section 4.2.1*. We thus hope to show the necessity of providing a situational classification functionality for on-line stack evolution.

6.2.1 Detecting Changes in the Network Conditions

We intend to verify that the population selector functionality we implemented has a positive impact on the adaptation process, i.e. whether our approach of utilising multiple independent populations and the selection between them based on the environmental or traffic characteristics benefits the system overall. For this purpose we devised a scenario in which the situational classifier is statically configured to utilise only fixed and reliably measurable decision criteria[3].

For this purpose we constructed an – inarguably somewhat synthetic – scenario of multi-factor cost optimisation, which is based on the following considerations: Mobile devices, such as laptops or smart phones, are often repeatedly employed in different locations, either connected to mains power or running on battery, and attached to different networks. Here we consider two different types of costs. Firstly, the monetary cost of communication over the wireless link is often based on the transmitted data volume. Secondly, extensive processing can quickly drain the limited battery power on

[3] While we implemented several automatic classification approaches such as k-means, we consider the actual details of on-line network classification out of scope for our current work and therefore (apart from the introduction in *Section 4.2.2*) assume that a reliable classification scheme is in place.

such devices. Thus our optimisation goal changes depending on the type of the power supply and the egress interface the device is using. If it is powered by battery, we try to reduce computational overhead. If it is communicating via WLAN, we try to reduce the amount of data transferred. If both conditions are true we let the system balance both goals. Additionally the system has to be able to fill its normal operational demands, i.e. data transmission. Thus we further instruct the system to minimize transmission loss.

One possibility for reducing the amount of transmitted data is to utilise data compression or to choose low-overhead protocols. The second cost factor, i.e. power consumption, is however negatively influenced by the use of data compression algorithms as they are often computationally expensive. The optimal point of operations is not obvious and depends on the weight of the two cost factors in the fitness function. But these weights obviously depend on whether the system is battery-powered and whether it is using the wireless interface.

Based on the aforementioned considerations, we assumed that the optimal stack configuration uses no compression at all when connected via Ethernet and on battery power, and the most effective compression method when connected via WLAN and the main power supply. The probable results for the other two cases were less obvious, due to the trade-off between the (power) costs and (monetary) benefits of compression.

We consequently selected the type of the egress interface and of the power supply as decision criteria for the classification process. Both criteria are easily detectable, as the necessary data is provided by most operating systems and offer the benefit of precision. After all, we do not intend to explore whether a particular classification methodology is effective, but whether our mid-term adaptation logic is effective when using a sufficiently precise classification method. We configured the classifier such that it selects between four populations based on the link state of the wired network (interface up / down) and the state of the connection to the main power supply (connected / disconnected).

Stack Modules

Again we provided the implementations of IPv4, Ethernet, and UDP, the configurable modules for TCP, DCCP, and the Codec module, in the same manner as introduced in *Section 6.1.3*. With respect to power consumption, the most effective algorithm, DEFLATE, incurs the highest calculation overhead and thus cost, which further increases the higher its compression factor is set. No compression at all is naturally the cheapest operation, followed by RLE, a method that was very popular on 8-bit microcomputers in the early

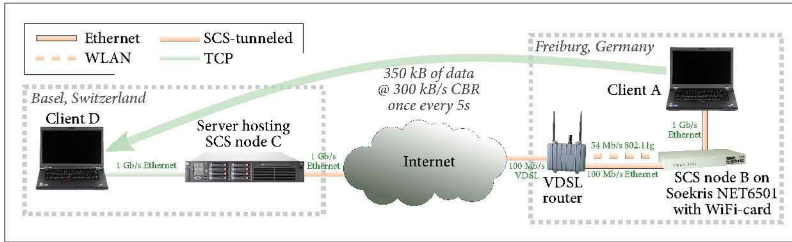


Figure 6.16: *The simulation environment is again modelled after the physical layout of out test-bed.*

1980s because of its low computation costs. On the transport layer UDP is the cheapest option, as it consists of a very basic multiplexing functionality and the calculation of the Internet checksum.

Network Topology & Traffic Characteristics

For this experiment we scheduled Client A, shown in *Figure 6.16*, to once every 5 s download a 350 kB file consisting of concatenated HTML pages from the web server hosted on Client D. As in the previous experiments, the focus of our experiments is the stack composition system located on SCS node B, which is connected via both WLAN and Ethernet to a VDSL router. We scheduled one of its network links to the router to be disabled and the other to become enabled every 60 s. Additionally, we simulated a switch between mains and battery power, also at the same period, but delayed by 30 s.

Fitness Function

As described above, we designed the fitness function such that it optimises the sum of the communication and power costs:

$$F = \bar{R} \cdot b_{0.5}(\bar{M}) \cdot b_{0.5}(\bar{E}) \cdot b_{0.25}(\bar{D}),$$

where the reception quality measure $\bar{R}$ denotes the ratio of application-level data that arrives at the recipient to the total data sent, and the delay measure $\bar{D}$ the ratio of delay to round-trip-time, as defined for scenario 1A. The transmission cost measure $\bar{M}$ calculates the monetary cost of transmission based on the amount of data sent on the physical layer and the interface type. The energy cost measure $\bar{E}$ is calculated based on the CPU usage C and the

power supply type.

$$\bar{M} = 1 - 2^{-log_{10}\left(1+\frac{S_P}{S_0}\right)} \cdot \begin{cases} 1 & \text{if using the wireless network} \\ 0.001 & \text{otherwise} \end{cases}$$

$$\bar{E} = 1 - 2^{-log_{10}\left(1+\frac{C}{C_0}\right)} \cdot \begin{cases} 1 & \text{if battery-powered} \\ 0.001 & \text{otherwise,} \end{cases}$$

where $C_0 = 3000$, $S_0 = 10^4$ are scaling factors derived through experimentation.

Experimentation

To explore whether our mid-term adaptation logic is effective, we performed two different experiments. In the 1st experiment, we used four populations between which the matrix-based classifier selects based the interface and power supply types, in the 2nd experiment the population selector logic was disabled. We executed 200 simulation runs both for the four-population and the one-population case. We synchronised the length (30 s) and start of the sub-trials with the interface and power supply switches. Since we scheduled one transfer every 5 s, we assumed that one sub-trial, i.e. 6 individual transfers, would offer sufficient normalization for our purposes.

As evolution logic we used the 1st configuration of the Composition Tree Search as introduced in the previous scenario. We then measured how the one generic stack configuration evolved in the one-population case fared in comparison to the specialised stacks evolved within the individual populations used for the four-population case.

Since we wish to verify our expectation that the population selector positively influences the overall adaptation process, we compare the individual best stack configuration per population that evolved for the four-population case with the one best stack configuration of the one-population case. Before performing these experiments, we assumed that in the first case the system would develop four distinct specialised stacks, each suited for the particular conditions under which it evolved, whereas the second case would result in one generic stack that performs reasonable well under for all tested situations. While this assumption turned out to be wrong, we still consider the experiment itself successful, as detailed below.

Results

As verification for our assumptions regarding the benefit of the mid-term adaptation logic we explored whether a noticeable difference between the

best stacks developed in each population in the four-population case existed, and whether these stacks exhibited better performance than the generic stack found in the one-population case.

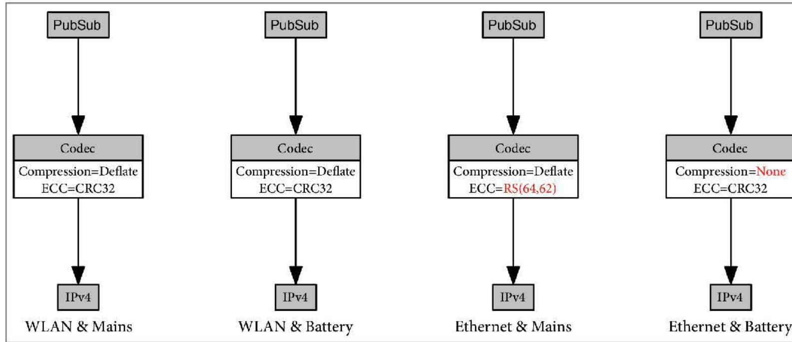


Figure 6.17: *The best stack configurations found for each situation in the four population case.*

The Best Stack Configurations We first verified that the stacks that evolved in the individual populations, i.e. situations, for the four-population experiment exhibited a different distribution w.r.t. to the utilised transport protocol, compression method, and error correction facility. The best stack found for all four populations after 25 generations are shown in *Figure 6.17*. When looking at the best-performing 20% (i.e. 50 experiment runs) for each of the four situations, the distribution was as follows.

When using the WLAN, DEFLATE compression was always used, as we had expected, using a compression level between 5 and 9. The same was also true for communication over the Ethernet when on mains power. Only on when running on battery and connected via Ethernet, compression was disabled in 38 cases, and DEFLATE used in the remaining 12. Here the compression level was distributed between 1 and 3[*].

When communicating via Ethernet and running on battery, the system directly used the Codec module without any transport protocol in 27 experiment runs, and UDP in 23. For WLAN and battery power, the transport protocols were bypassed in 30 experiment runs, and UDP used in the remaining 20 cases. When on mains power, all 50 stacks used the Codec module directly, independent of whether the WLAN or Ethernet was used. We assume

[*] These modes do not use lazy match evaluation and are therefore faster than levels 4 and above.

that the relatively low overhead added by UDP had too little impact on the overall fitness compared to the selected compression method when running on battery power, especially considering that the processor time needed for packet handling fluctuated considerably, as apparent in *Figure 6.18*.

Error correction was always disabled when running on battery power, since it added measurable computational and some data overhead. When on mains power and using the WLAN, RS(64,62) was used in 8, RS(64,60) in 2, and CRC32 instead of error correction in the remaining 40 cases. Only when running on mains power and communicating via Ethernet error correction became prevalent. Here RS(64,62) was used in 30, RS(64,60) in 6, and CRC32 in 14 experiment runs.

We then checked that the bests stacks found per population exhibited equal or better performance for the particular set of conditions the were developed under than those found for the other populations. This was always the case. For example, the best stack configuration for the Ethernet and battery case, which used no compression, performed worse than the best stack found for WLAN plus mains power situation (fitness 0.0551 vs. 0.5545). When using the best stack evolved for the Ethernet and mains case, which uses RS(64,62) error correction, under these conditions, it achieved a fitness of 0.5544, which is only slightly lower than not using error correction, and thus also explains why Reed-Solomon was utilised by 10 of the 50 best-performing stacks under these conditions.

Considering these results, we had assumed that a generic stack developed for the one-population case, i.e. without classification would use the Deflate compression and CRC32, omitting the transport layer. Our analysis of the results however showed that this was not the case. Instead the fitness of the best stack in the population exhibited no improvement over time at all, as described below.

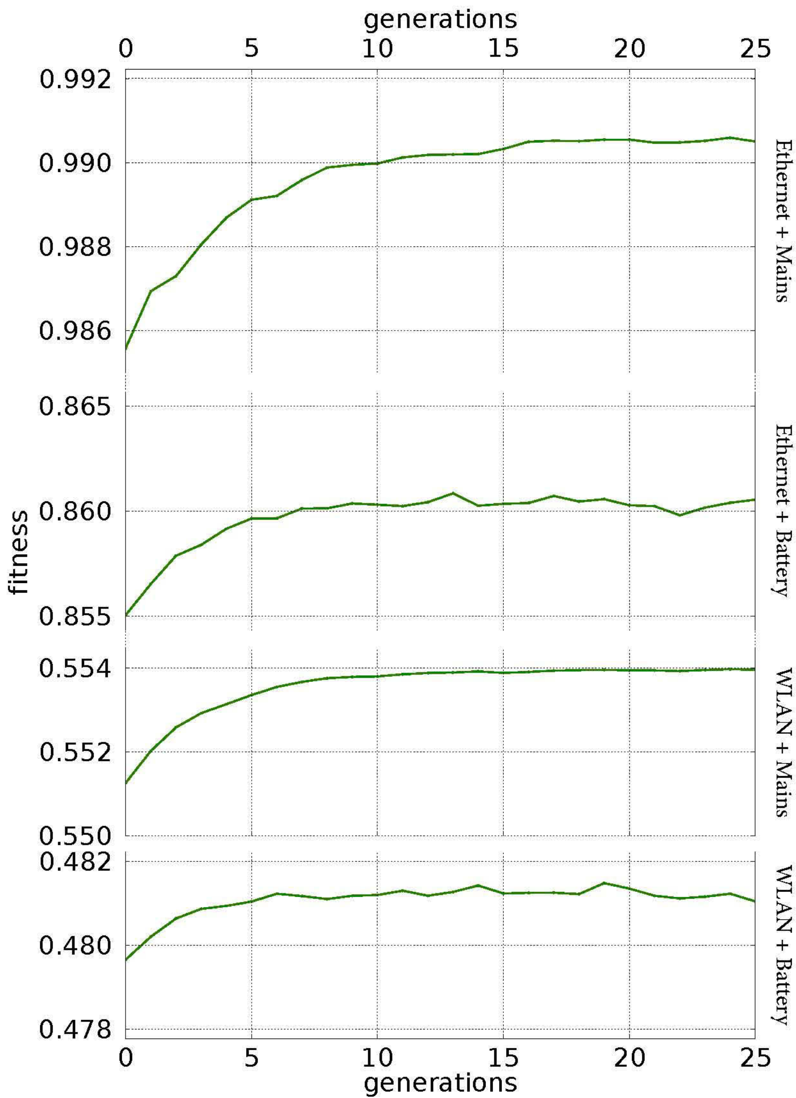


Figure 6.18: *Since the decision criteria for clustering in the current case are optimal, the fitness for the individual populations also improves as if the scenario would not change.*

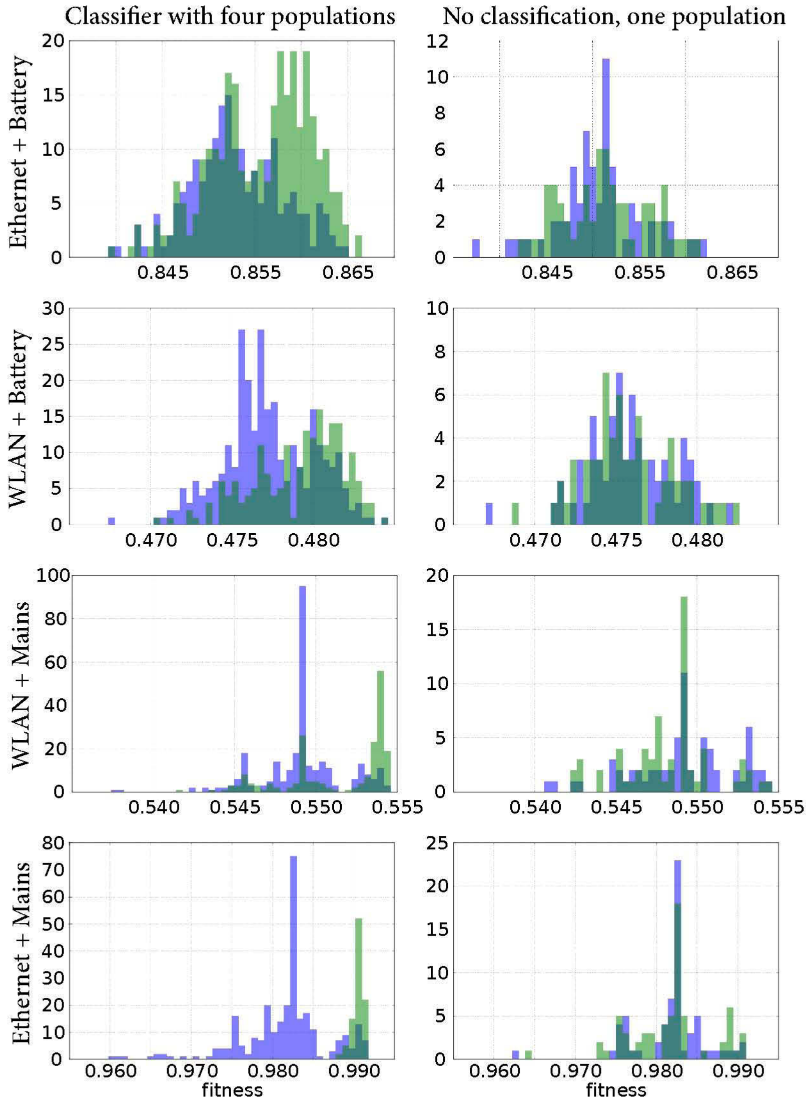


Figure 6.19: *The improvement of the overall stack fitness becomes especially apparent when comparing the histogram of the randomized stacks at the start of the experiments (blue) to the result at the end (green). When using multiple populations (left) and correct classification, the average stack fitness after 25 generations noticeably improves. With only one population (right) and thus without classification, no progress is apparent.*

Performance of the Evolution Logic In *Figure 6.18* we show how the fitness for the individual populations improved over time for the four-population case. As the population selector was able to correctly discern between the different situations the evolution quality and speed resembles the results we got when performing four individual experiments for each of these situations. As we had assumed, the overall adaptation speed was proportional to the number of populations, which again highlights the importance of choosing the right number of classification criteria: If n redundant criteria are defined the overall adaptation speed gets n-times longer, thus a low number of populations is preferable. Choosing too few criteria, however, is unexpectedly detrimental to the adaptation speed: As shown in *Figure 6.19*, the omission of the classification process during the one-population case resulted in a dramatic failure of the stack composition system to improve the performance at all. This is probably due to the fact that a closer-to-optimal stack configuration produces worse performance measures than an inferior configuration, if it happens to be trialled more commonly during worse conditions. A higher number of sub-trials might reduce the random effects caused by scheduling experiments under different conditions, but the necessary number of sub-trials would be rather high, considering that they e.g. a close-to-optimal stack when on battery power and communicating via WLAN has a fitness below 0.5, but when on mains power and connected via Ethernet exhibits a fitness above 0.9: For every sub-trial out of e.g. 10 is performed under such worse conditions the average fitness gets reduced by 0.05. Thus it is probably more efficient to over-provision the number of classification criteria than to under-provision and rely on performing more sub-trials.

Conclusion With this scenario we showed that the use of multiple populations, which are individually evolved for a particular set of environmental and traffic conditions can not only be beneficial for the overall system performance, but – in some situations – even vital, as it helps to stabilise and normalise the experimentation environment. In this scenario the autonomous long-term adaptation encompassed in the evolution engine was only possible because of the precise classification of the situation and selection of a different population whenever the situation changed. Without this logic, no improvement was noticeable. The difference in stack configurations was not as pronounced as we had hoped when we designed the scenario, apparently because the calculation cost for the compression algorithms was too low to sufficiently impact the fitness value, especially compared to the cost of transmission. But nevertheless, the performance difference between a generic and

the specialised stacks became apparent. The different distribution of stack configurations for each of the situations, the difference in relative fitness between these, as well as the different position of the optimum as indicated by the direction of evolution, also strongly indicate that situational classification is useful and maybe even necessary for autonomous stack evolution.

6.3 General Conclusions Derived from the Experiments

In this chapter we presented some of our experiments, which when taken together act as a proof of concept for the long-term evolution approach, the mid-term population selection logic, and for the stack composition system applicability for some autonomous stack optimisation tasks. While several aspects of the design remain untested – mostly due to the sheer size of the problem space – we offered first insights into its abilities.

6.3.1 Long-Term Stack Evolution

By applying the evolution engine to different adaptation problems, we showed that it is capable of improving the network stack on its own towards a goal defined by means of a fitness function, provided that the experimentation environment stays stable enough. In particular our Composition Tree Search algorithm, which we developed especially for this task, is a promising candidate algorithm for this purpose, as it showed the best overall performance in our experiments.

These experiments also indicate that autonomous reconfiguration based on run-time decision is worth further exploration: We showed that the performance of the algorithms we designed and applied for this purpose depends on the particular problem, including the run-time conditions, and that our approach of providing multiple algorithms, which can be selected and configured at run-time is therefore sensible. Our experiments also confirm that the adaptation speed of the long-term adaptation facilities on their own is too slow for most practical applications. We therefore see the need for additional mid- and short-term adaptation methods confirmed.

6.3.2 Mid-Term Adaptation

We further showed that a mid-term adaptation logic, which saves the progress of and switches between different stored sets of long-term evolution state, can not only increase the speed of adaptation to already known conditions, but also make experimental trial-and-error adaptation itself

easier: It separates experiments into different groups depending on the conditions, and thus inherently reduces the variety of conditions encountered during experiments that are compared with each other. The scenario we presented for this purpose shows very clearly that only because we provided multiple populations in which different stacks were evolved for each of the different environmental conditions and application requirements, the system was able to adapt a set of sensible stack configurations and offer close-to-optimal performance in the given scenario. Without this facility in place, the stack composition system completely and thoroughly failed to adapt. In the future we plan to investigate this problem further, and especially wish to investigate whether truly autonomous classification based on criteria learned at run-time is feasible.

6.3.3 Applicability

Lastly we showed that our design can be integrated with current operating system and act as a replacement for the network stack deployed there. Whereas our implementation in currently located in user-space, the experience we gained confirms that current router-class hardware provides the necessary processing power for hosting our stack composition system, and that our design does not cause a significant impact on stack performance: Only when actual re-configuration and evolution is performed, was an increased delay noticeable, and even then it was infrequent and in the low millisecond range. For practical application, however, further work is still needed.

Chapter 7

Discussion

With this chapter we conclude the description of our research related to autonomous on-line network stack evolution. We begin by summarising the content of the previous chapters and continue with a brief recapitulation of our efforts, in which we describe how much of our vision we have achieved as well as the limitations of our work. We also provide an outlook towards the future, in which we focus on those areas of research we consider the most fruitful and worthwhile.

7.1 Thesis Summary

After a brief introduction of the problem space and our work in the first chapter, we detailed the motivation for our research in chapter two. Here we presented our view of the problems and limitations of the current Internet and proposed replacement architectures and stated the need for a situation-aware adaptation logic based on autonomous exploration and performance measurements. We then formulated the requirements and our vision for a stack composition system that encompasses this logic and continued by stating our contributions towards achieving this goal.

In the third chapter we introduced related research in the fields most closely connected with our work, as well as some of the technologies required for or affected by the stack composition system. Here we focused on the features pertaining to the achievement of autonomy, i.e. the collection and analysis of information needed to realise situational awareness, the learning and decision process which based on this information decides how to adapt the system, and lastly the actions that the system may take to influence itself

and its environment.

The fourth chapter describes our design rationale and the system architecture we devised and implemented for the stack composition system. We detailed the requirements the system has to fulfil and based on these developed the architecture for the compositional framework, which constructs the stack out of micro-protocol modules, controls and steers the system's operations as well as the evolution process, and detailed the components and technologies encompassed in the stack composition system.

In chapter five we detailed the evolution engine, i.e. the component that learns and decides how to adapt the stack to the network and traffic environment. We detailed the mechanisms and algorithms employed for steering the short-term, mid-term, and long-term adaptation process. We described the long-term evolution process which is realised by means of machine-learning algorithms, e.g. an Evolutionary Algorithm or the Composition Tree Search we developed, discussed the mid-term adaptation which is governed by the classification and population selection mechanism which selects a different pool of candidate stack configurations whenever the conditions change significantly, and introduced the short-term adaptation process realised through special micro-protocols that are emplaced within the stack by the long-term adaptation logic, but which operate on a far finer time scale.

The sixth chapter contains our discussion of the difficulties pertaining the accurate gathering and assessment of information and measurements from within a live system in the presence of noise, while the system itself influences and is influenced by the conditions in the network. We presented and discussed our strategies and countermeasures which enable the system to effectively and accurately gather and assess this information. We further described the methods we employ for calculating the stacks' utility and for scheduling the experiments, as these directly relate to the aforementioned problems.

Chapter seven details a selection of the experiments we performed to validate the system's functionality and to evaluate its effectiveness, i.e. the quality of the stacks evolved measured by means of their utility in relation to the time elapsed. Here we focused especially on the long-term evolution process, as it has the highest impact on the system's overall utility, followed by a brief evaluation of the benefit offered by the mid-term adaptation layer.

This eighth and final chapter concludes our thesis and contains a summary of our work, the description and discussion of our contributions and achievements, as well as the limitation of the work, followed by an outlook for future work needed to achieve our vision.

7.2 Contributions, Limitations & Future Work

In *Section 1.4* we introduced our contributions to the field of autonomous networking in general and autonomous network stack evolution in particular. We now look back at our statements and analyse whether and how well we managed to realise them. For this purpose we give a brief overview of our work pertaining to each of the contributions, followed by a comparison to related work, and a discussion of the benefits and limitations of our solution.

7.2.1 Architecture

Our first contribution relates to the design and implementation of a framework which enables autonomous stack evolution. We presented the comprehensive design of this framework in *Chapter 3*, implemented this architecture as detailed in *Section B*, and assessed its abilities in *Chapter 6*.

We see our research as one step into the direction of realising the knowledge plane[63] for the purpose of autonomic network stack composition and configuration based on trial-and-error experimentation. We do however deviate from this concept in one important point: Our system does not support collaboration between different network nodes, the adaptation process itself is intrinsically limited to the local node itself. But while we do not provide a supervisory entity to control the evolution process, we do implicitly encourage cooperative evolution by means of remote satisfaction measures and their inclusion in the local fitness calculation.

As we were able to utilise our framework as a replacement for the operating system stack, route the network traffic through it, and perform our experiments as intended, we consider our design and implementation efforts successful. Our work is however pending a more thorough analysis and implementation of further protocols and services, thus no final statement can made as of yet. We can however say that the programmatic module interaction greatly simplified our implementation efforts since it allowed us to use common programming paradigms. Our constraint ontology further helped us to prevent the generation of invalid stack compositions.

So far we most thoroughly used and tested our stack composition system for the configuration and composition of (micro-)protocols on the internet and transport layers of the protocol stack of stand-alone communication endpoints. We consider the system well-suited for this purpose, under the conditions we discuss further below. We do also assume that the approach is similarly useful for several related problems. For example, the configuration of application-level options, e.g. the feature selection for VoIP can easily be integrated into the system, provided that sufficient user feedback

is available to determine the impact of the selected option on the perceived utility. Similarly to the usage of optional TCP features that can be negotiated between communication endpoints, the per-call configuration of VoIP connections can be determined by the calling node on its – with the possibility of falling back to the settings required by the callee, if it should not support the feature(s) in question. This methodology resembles our approach for initiator-defined composition introduced in *Section 3.7.5*: Since the optimal configuration per node is therefore independent of other nodes' configurations, the optimisation process should be straight-forward.

The configuration and selection of distributed, e.g. routing, protocols may also be feasible with our approach under controlled conditions, as we intend to evaluate in the future: If all nodes in the network are guaranteed to provide a common base of supported operations, e.g. requests for link state information, and can handle each other's status update messages, the individual optimisation of the routing protocols' settings on a per-node basis should be feasible. After all, the routing tables themselves are usually managed by each node on its own. Guaranteeing comparable experimentation environments under such conditions is however probably far more challenging, as already mentioned in *Section 5.2*.

While we designed our system with the demands of future networks in mind, our approach might already be useful for some applications on the current Internet when no support from communication partners is needed. For example, we consider deploying our system on a home network router to optimise the packet queuing priorities and bandwidth limits per packet class for the purpose of optimise throughput and delay. Here the system would implement changes to the firewall queuing rules, measure the results of these changes over the period of one day, and adapt the configuration appropriately afterwards – a process we so far performed by hand every time we moved to a new Internet provider or apartment.

For several other application areas, however, the autonomic evolution of the correct configuration or composition is probably unrealistic. The security of cryptographic protocols, for example, cannot be determined through trial-and-error experimentation. Considering the large amount of thoroughly reviewed encryption solutions that contained bugs or were based on misconceptions that rendered the actual security moot, and that these flaws were noticed only after a sometimes very long time,[143,144] we conclude that our autonomous approach is unsuited for this particular application area.

In general, we expect that a shared common code base and flexible feature negotiation are vital for achieving good performance in a distributed environment. A heterogeneous (network stack) environment can proba-

bly only function close to optimally, if the entities therein are sufficiently tolerant, i.e. can cater to the needs of their communication partners: The independent selection of the protocols and features on a per-node or even per-flow basis depends on support for the same functionality by the communication partners. Even though backwards-compatibility and support for fall-back options in case of missing features is likely unavoidable, the flexibility of the adaptation process is severely constrained if a majority of a node's communication partners do not support the features needed to improve its utility. Just like a multi-lingual society requires good language education, our approach depends on a sufficiently large capability of the nodes to realise optimal communication between them.

Our approach does not preclude incremental deployment of functionality or even communication between incompatible subnets. Incompatible addressing schemes, for example, can prevent communications between nodes, but the fall-back to a common baseline scheme can guarantee an operational network while a newer and better scheme is deployed incrementally on the infrastructure. Once the more advanced scheme's penetration of the network is sufficient, the autonomously adapting nodes can then decide to switch – provided the utility is actually higher. Likewise, translation services, e.g. the Interstitial Functions of Plutarch,[72] can help solve the problems of communication among heterogeneously-configured nodes or networks, and thus eliminate the need to distribute and provide a large code base on every node.

7.2.2 Cognitive & Learning Facilities

Our second contribution concerns the algorithmic side of the long-term adaptation process, which enables the evolution engine to direct the adaptation of the network stack towards a goal state defined by means of an utility function. While our results indicate that our approach can autonomously adapt the network such as to optimise the perceived utility, we also noticed that the performance of the search algorithms strongly depends on the particular problem space, as well as the run-time conditions, and that for optimal performance the best-perfoming evolution logic thus has to be selected at run-time.

We designed multiple algorithms for this purpose as detailed in *Section 4.1*, which resemble and were modelled after common constraint optimisation and search algorithms. We explored how these algorithms perform in several application scenarios, i.e. we measured the fitness of the best stack evolved per generation and compared the adaptation speed and quality for different algorithms and parametrisations of these.

The scenarios we explored use the current base of widely deployed networking protocols, for the reasons we described in *Chapter 6*. We do however assume that future application areas of autonomous stack evolution will be more complex and higher dimensional: Current protocols are devised to exhibit sufficient performance for many application areas under various conditions by default. They usually provide few possibilities for fine-tuning, because finding the best possible parameter configuration by hand is hard and requires expert knowledge.[352] If autonomous adaptation should become wide-spread, this is likely to change. Since no expensive hand-optimisation is needed, we expect future protocols to provide far more possibilities for parametrisation, and also more specialised protocols to appear and to actually be deployed. To compensate for the lack of configurability in the existing protocol base, we decided to include our own extensions for existing protocols, e.g. by means of proprietary TCP options, in the test scenarios and also to develop our own highly-configurable protocols, e.g. for compression or error correction. The resulting scenarios thus offer higher complexity than the current Internet protocols on their own would allow for, but still retain sufficient realism.

The selections of algorithm designs was exceptionally difficult, as the problems we intend to apply them to are very diverse. Whereas many algorithms for related application areas are discussed in the literature, the conclusions drawn there are either not applicable to our problem or too general in nature. Consequently, we did not possess sufficient knowledge to decide which kind of search algorithm was most most appropriate for our purposes, and therefore selected some of the common model-free approaches for constraint optimisation. Since the fitness landscape is defined by the – at design time unknown – fitness function, we chose evolutionary algorithms, which are known to work comparatively well for complex landscapes, as mixing techniques such as crossover and mutation helps to move the search away from problem areas like e.g. local maxima. Similar approaches have been successfully applied for e.g. QoS-based[385] or shortest path routing.[6] Our experiments described in *Section 6* however show that our Evolutionary Algorithm implementation is – at least for the necessarily somewhat simple scenarios we presented – inferior to random probing. Whereas we were aware that no possible algorithm can always outperform even the simplistic random search,[381] the general properties of Evolutionary Algorithms initially made them seem very promising and led to our choice. The ineffectiveness of the algorithm for our optimisation problems is aggravated by the rather small population size on which we let the algorithm operate, and which further highlights the need for specifically-designed algorithms: Our need to keep the number of candidate stacks and thus experiments low is signifi-

cantly different from the conditions for which these algorithms were developed. Yet we still expect this algorithm to exhibit better performance for more complex and higher-dimensional optimisation problems, e.g. the optimisation problem likely posed by future network stacks as discussed above.

The Composition Tree Search algorithm, which we specifically developed for the problem at hand showed superior performance, as it outperformed all other algorithms we had investigated, and especially the Rapidly-Exploring Random Tree which inspired its design. The Rapidly-Exploring Random Tree in fact consistently failed to improve the stack composition and thus was excluded from our experiment description. The importance of correct parametrisation also became apparent in our experiments, and seems to depend on the problem set – but a definitive conclusion would again be premature. Dynamic re-parametrisation also showed promising results, but it is still unclear when and how to re-configure the algorithm at run-time. Other algorithms, such as the relatively recent Cuckoo Search,[388] also deserve consideration, and are considered for future work.

Since the probability distribution for the problem class we investigate is unknown at design time, i.e. the evolution of stack configurations, depends to a large extend on factors such as utility definition, available modules and their utility when applied to actual traffic, any preliminary exclusion of possible algorithms from our design would have been counter-productive: We do not even know the dimensionality of the search space a priori, as it depends on available modules at run-time. Likewise, the distribution of the optimal or close-to-optimal configurations across this space is unknown, and we *cannot know* in advance what the future network protocols or user demands will be like. We came to the conclusion that multiple algorithms have to be provided by our stack composition system, and that the correct choice and parametrisation of this algorithm can only be determined at deployment or at run-time. Our design therefore explicitly allows for on-line selection and configuration of the employed algorithm.

Currently our design allows for the specification of the fitness function at run-time, based on sensor measurements provided by the system and the modules that constitute the stack. Lee's research into user-guided utility measures[211,212] seems very promising for future work, as the addition of such a facility might enable our system to "transcend" the sometimes difficult problem of defining an utility function which explicitly encodes the users' intentions.

7.2.3 Situational Awareness & Knowledge Base

Our third contribution concerns the mid-term adaptation logic, i.e. the population selection mechanism described in *Section 4.2*, as well as the algorithms used to realise this functionality. Here we showed that the functionality offered by this layer is required and that our approach increases the potential application areas to which our long-term evolution approach can be applied. We further witnessed that appropriate configuration by hand is currently needed for this method to be effective.

We implemented a matrix-based method for classification based on user-defined criteria, as well as a k-means based clustering algorithm. We were able to experimentally show the *need* for such functionality, and the ability of our framework to host the necessary algorithms. For the scenario we explored in *Section 6.2.1*, autonomic adaptation does only occur when the network situation is correctly classified and distinct populations are selected for different conditions. The matrix-based method introduced in *Section 4.2.1* was sufficient for this task, which seems to indicate that this approach is sensible when reliable classification criteria can be derived by the administrator. With concern to autonomic selection of criteria, however, further and more thorough research is definitely needed. We sketched a method for dynamic determination of the classification criteria and their weighing in *Section 4.2.3*. So far we only performed initial experiments for a very low number (≤ 6) of possible inputs, which we consider inconclusive[61] due to the difficulty of realising a realistic scenario either within the simulator or our test-bed. We plan to further investigate the possibilities of automatic classification in the future, and consider exploring other algorithms e.g. ANNs as well, which have been successfully employed for similar purposes.[356]

The method we proposed is however still only a sketch and not fully autonomous as of yet. So far we need to specify the necessary baseline stacks by hand, since we do not know how to reliably let the system find usable compositions at run-time. Our approach further requires an interpolatable fitness landscape, i.e. the fitness of a baseline stack under an untested situation needs to be approximately derivable from surrounding situations. We also do not know what metrics are adequate for measuring proximity between situations. Furthermore, before the system is able to start its autonomous classification process, it has to apply and test the baseline stacks for a sufficient subset of the possible conditions.

Our mid-term adaptation logic in its current state is thus not fully autonomous, as it requires operator knowledge to be effectively configured. But as we showed experimentally, it enables adaptation in situations were

[61] These experiments are therefore not included in this document.

the long-term evolution logic on its own would be insufficient. We consider the necessary efforts for proper configuration at run-time a small price to pay for the possibility of extending the problem scope to which our long-term evolution logic can be applied.

7.2.4 Information Gathering & Assessment

Our fourth and final contribution encompasses the facilities for autonomous information gathering and assessment, which we detailed in *Chapter 5*. These measures were sufficient to normalise the measurements and derive comparable fitness measures during our experiments described in *Chapter 6*. More exhaustive experimental evaluation of our approach is however necessary. Furthermore, our architecture includes facilities for collaborative fitness calculation based on sensor reports and a generic satisfaction, and thus the possibility for remote entities to pro-actively influence each other's experimentation and stack evolution. We utilised these reports throughout our experiments e.g. to derive the remote reception quality for UDP data transfers. We plan to perform further experiments to assess the possibilities of explicitly influencing a remote node's experiment scheduling and its adaptation, as well as collaborative adaptation towards a common goal.

7.3 Concluding Remarks

For our research we developed a framework for network stack evolution which can be used for experimentation and exploration with different stack compositions and to test new protocols. Our research so far only covers some of the necessary aspects of a complete replacement architecture for future networks. During our – far from exhaustive – experimentation, we witnessed the ability of the system to reliably arrive at usable stack configurations which performed better than a generic, i.e. non-optimised stack. The algorithms we developed for autonomous evolution, information gathering and classification still offer much room for improvement, but were already usable as our experiments with a limited set of optimisation problems, and may serve as a basis for further research and optimisation. In particular when the environmental and traffic conditions remain stable for a sufficiently long period of time and fast adaptation is not needed, the evolution engine on its own can already be used to adapt the network stack, even when no background knowledge about the situation in which it is applied is give, as only a suitable definition of fitness and a specification of the available stack modules is needed. If the traffic or environment changes, but an adequate

measure for these changes is known, the mid-term classification and population selection approach may be used to let the system adapt to multiple different sets of conditions at once, and switch almost instantly between these sets.

While we consider the results so far promising and as warranting further research, many open questions remain. Here we consider further research of the situational classification and collaborative multi-node evolution, as well as the development of a larger code base of micro-protocols for experimentation, most urgent. Especially the latter problem, i.e. the lack of suitable micro-protocols hindered our experimentation efforts as we had to implement all protocols ourselves, which severely slowed down our progress. Apart from the need for sufficient development time, no fundamental problems became apparent during our research, which is why we are confident that further evaluation of our approach is worthwhile and hope that our research may lay a foundation for a future autonomous stack composition that is usable in practise. But until such a system becomes reality, a lot of work still remains.

Appendix A

Protocol Design Requirements

Most current network protocols have been designed with little or no consideration of the effects their operations can have on other protocols and how they might interact with future protocols, which led to numerous problems and consequent calls to design for change.[64] For the stack composition system the possible detrimental effects a protocol can have on the operation and performance of other protocols or the stack as a whole are likely grave. Many protocols have only been designed to work within the rather static TCP/IP protocol environment. Stack configurations obviously do not remain static in our system, and can arbitrarily change at any point in time, and thus induce unpredictable effects in the network. And since our concept specifies that the initiator of a communication flow defines the stack configuration for that flow (see *Section 3.7.5*), multiple protocols and stack configurations can even be active and in use at the same time, and switched between.

Furthermore, to enable effective stack composition we require support for features that current protocols were not designed for. For example, most protocols available in the Internet stack were intrinsically designed with permanent reachability in mind, so that e.g. TCP connections are known to time out easily under intermittent connectivity.[illegible] Since we intend protocols to be exchangeable in the middle of communications without interrupting ongoing connections (see *Section 3.2*), a standard-conforming TCP is not sufficient and had to be extended to be able to handle being switched out and back in at arbitrary times.

Luckily, some of the negative effects a protocol can have on the network can be mended through cooperation between nodes and the use of adequate

[illegible] A problem which is addressed by DTN and opportunistic networking.

fitness definitions (see *Section 5.2*): For example, TCP-friendliness might be achieved by penalising unfriendly behaviour in the fitness assessment. However such an approach is not universal and adds unnecessary complexity to the configuration instead of reducing it as we intend. We therefore propose requirements for protocol design in this section which are intended to curb possible hindrances for stack composition. Even though we do not require conformance to most of the requirements or recommendations we introduce below, adhering to them increases the possibilities the stack composition system has for composition and configuration and thus might – at least in theory – lead to higher utility.

Even though many of the requirements or suggestions given in this section are likely to cause additional development overhead, we expect adherence to be beneficial even when the protocol is employed outside of a stack composition environment: At least some of the problems that burden current networks could have been ameliorated or even prevented, if the protocol design had conformed to the guidelines, a notion we try to clarify by means of the examples given below.

A.1 Monolithic and Flexible Protocol Design

We require protocol implementations to be monolithic with respect to dependencies on other protocols, in particular they have to be oblivious of as much of the characteristics of other protocols as possible, since intrinsic dependencies impede the run-time re-configuration of the stack.[358] The standards that define TCP and UDP, for example, do not conform to this requirement, as they both enforce the inclusion of the underlay's IPv4 or IPv6 addresses in the checksum calculation. Such a dependency on a specific underlay naturally precludes running these protocols on top of at-design-time-unknown protocols, as explicit support for these has to be added to the implementation. And even worse, IP-level routers need to understand both protocols as well to be able to re-calculate their checksums as the replace addresses when crossing network boundaries. Our implementations of TCP and UDP therefore intentionally deviate for the standard and omit to include the underlay addresses whenever the underlay is not IP.

Protocol design shall thus avoid unnecessary dependencies on other protocols, and all necessary dependencies should be specified and implemented in such a way that other modules which offer comparable functionality can easily replaced for the ones intended at design time (see *Section A.7* below).

A.2 Configurability

Protocol designers often take decisions which are based on assumptions about the operation environment or traffic conditions. For example, the default time-out or the size of sequence numbers for TCP are based on assumptions made in the 1970s, and naturally do not reflect the drastically different nature of the networks we are experiencing today: Connections were defined to time out after five minutes (see RFC 793[289]), window sizes were limited to 16-bit and sequence numbers can overflow within 17 seconds on Gigabit Ethernet (see RFC 1323[171]).

Such decisions should therefore ideally be taken at run-time, for example by the stack composition system whose reason of existence is to find the best configuration. While demanding total configurability is probably unrealistic – e.g. supporting multiple sequence number sizes would preclude some code optimisations – other settings, such as the default time-outs, can be changed without requiring modification of the implementation. In fact, TCP explicitly supports this feature from the beginning, and the setting can be changed through run-time knobs in most current operating systems. And just like the window size limit was later extended through the introduction of a scale factor (again see RFC 1323[171]) to reflect the increased bandwidth of modern networks, a protocol can be designed to be extendible from the beginning. When this is not done, such as in the case of the rigid and inextensible definition of 16-bit sequence numbers in TCP, using the protocol in different settings becomes difficult, in the aforementioned case PAWS, a workaround to protect against the effects of rapid counter overflows (also see RFC 1323[171]) became necessary.

A.3 Reliability

The next requirement is not specific to stack composition, and has to be considered essential for software architecture in general: Robust and reliable engineering. While protocol design almost always involves simulations, development of a reference implementation, and consequently debugging, the variety of the traffic and network conditions the protocol is exposed to in this phase is naturally limited. Our stack composition system, however, is comparable to a fuzzing tool: It non-deterministically, but indiscriminately, explores all possible stack configurations and thus often exposes hidden dependencies and insufficiently specified expectations by the designer or implementer.

We therefore require robust, resilient and reliable protocol design: All in-

trinsic dependencies and requirements a protocol might have on other protocols need to specified. We provide a specification ontology specifically designed for this task, such that the stack composition system does not accidentally connect a protocol to another one which does not offer required functionality (see *Section 3.4.2*). The specification for the protocol, however, has to be precise and comprehensive. In particular, hidden dependencies on other protocols, their behaviour, or the stack composition in general have to be avoided.

Even secure and resiliently developed protocols can, however, still contain programming errors or other bugs. No matter how careful the implementation and design process was checked, this is sadly unavoidable. The stack composition system therefore contains measures to detect and abort stack configurations that cause execution errors, as described in *Section 5.3.2*. However, this process is not, and cannot be, absolutely reliable, especially memory or data corruptions can cause unrecoverable harm. Therefore, thorough testing and debugging of protocols is vital.

A.4 Altruism

In a network environment which can be expected to be heterogeneous and thus contain multiple variants of the same protocol and multiple protocols of the same class, e.g. transport or routing, protocol design has to consider the effects their actions can have on other protocols even more thoroughly than would otherwise be necessary. As mentioned above, TCP-friendliness has only become a topic warranting research once the low throughput of TCP flows competing with UDP became apparent: The original protocol design for UDP intentionally left rate-control measures to the higher layers. The utter failure of this notion, which partially is due to application developers' ignorance of the problem or also lack of interest, consequently led to the development of TCP Friendly Rate Control (see RFC 5348 [114]). Naturally, not all negative influence is foreseeable or avoidable, but an analysis of possible effects on other network entities can help.

A.5 Co-operation

The stack composition system is modifying the stack at runtime while communications are ongoing. This process is – intentionally – inherently oblivious of protocol- and network-specific concepts like connections, flows, or network state in general. This implies that e.g. two connection-oriented transport protocols can be replaced with each other while connections are es-

tablished, unless constraints against this replacement are explicitly specified for the protocol modules. If both protocols are now implemented in such a way that they can utilise the same shared data structures – in which the connection-specific, but not protocol-specific, information is stored – then connections can be kept alive across stack modifications. Obviously, designing and implementing protocols in this way can be burdensome, but the possible benefits should be apparent. Protocol designers and implementers are therefore encouraged to store information in a generic way such that other protocols providing the same service can understand and utilise the information when possible. Another – yet easier to implement – example is a mechanism for RTT estimation. While the method of information gathering and formulae applied might differ, if the resulting assessment or the measurement data is stored in a shared structure, a newly instantiated method will not have to start from scratch, but can benefit from the knowledge obtained through the previous method. Naturally, this concept can complicate experimentation, as interactions between different protocol implementations can make effects that influence their performance more likely to occur. Not only can programming errors, such as faulty entries in a shared storage space, cause execution faults, but even correct operations can cause a burden for a following experiment, by e.g. filling up a shared send queue, but such problems can occur anyway and are taken care of by our experimentation methodology.

A.6 Restartable Protocols

Most common network protocols operate under the assumption that they are more or less permanently available. Except for some implementations of protocols which are explicitly designed for intermittent communications, implementers and designers naturally assume that the protocol itself is kept in memory and running. In the context of the stack composition system which modifies the stack at runtime this notion is obviously no longer valid. Protocols should therefore be implemented such that they can cleanly shut down and restart operations, e.g. through *micro-reboots*,[46,47] whenever signalled to do so by the stack composition system. A persistent storage space is provided by the system, in which arbitrary data can be kept even after a module instance has been shut down. Networking protocols, however, often[315] include state information pertaining to e.g. ongoing connections, which is shared across the network and might necessity signalling to the remote communication partner to shut down the connection within a predefined amount of time. Likewise, protocols should be implemented such that they allow re-configuration during operations.

A.7 Modularity

Transport protocols, to stay with the previous example, often include an error detection mechanism, e.g. the aforementioned checksums, which act as a safeguard against data corruption. These mechanisms, however, are utterly redundant if the underlying protocols guarantee reliable and incorruptible transmissions. In error-prone environments, however, they are likely to be insufficient. Such functionality should therefore either be omitted or externalised as an independent and generic service, for which the stack composition system can select the appropriate implementation.[355]

By separating common functionality such as error correction and detection mechanisms from the core implementation of the protocols, one can not only reduce redundancy, but also increase the possibilities for stack composition system to optimise stack operations, as monolithic designs might be easier to implement and test, but lack versatility.

For efficient stack composition, we therefore require the network stack to be highly modularised, i.e. protocols are to be split into more-or-less atomic modules whenever possible and sensible. Otherwise, if splitting functionality into independent modules is not advisable, decisions for selecting between different sub-functionalities within the module shall be made available to the composition system.

For example, protocol designers might separate the error detection and correction mechanism from transport protocol modules and instead provide hooks for the composition system to enable arbitrary methods at a place in the stack of its choice. The composition system can now prevent the addition of multiple, possibly redundant instantiations of CRC as currently in place in IP and TCP/UDP. Instead it might e.g. provide Reed-Solomon codes directly on top of the physical layer for lossy wireless links, and none at all for highly-reliable wired Ethernet links.

Appendix B

Implementation and Experimentation Environment

In this appendix we present additional details for both the user-space and the simulator-based implementation of the stack composition system, and introduce the micro-protocols and service modules we provide.

B.1 Implementation

As introduced in *Section 3.5.1*, the stack steering system is conceptionally located next to the operating system's stack, which it controls and modifies as it deems appropriate. While we realised this concept exactly as described before within the ns-3 simulator, for practical reasons we decided to deviate slightly from this concept for our standalone implementation. The necessary modification to the operating system kernel would not only have drastically increased the implementation and debugging overhead, but also restricted the possible deployment to the specific kernel version we would develop the system for.[1] We therefore decided to implement our standalone system in user-space instead, as detailed below.

B.1.1 Stand-alone User-space Implementation

As shown in *Figure B.1*, the user-space implementation interacts with the operating system and applications through several complimentary interfaces.

[1] Especially the Linux kernel is infamous for introducing incompatible changes to supposedly stable APIs and data structures even between minor versions.

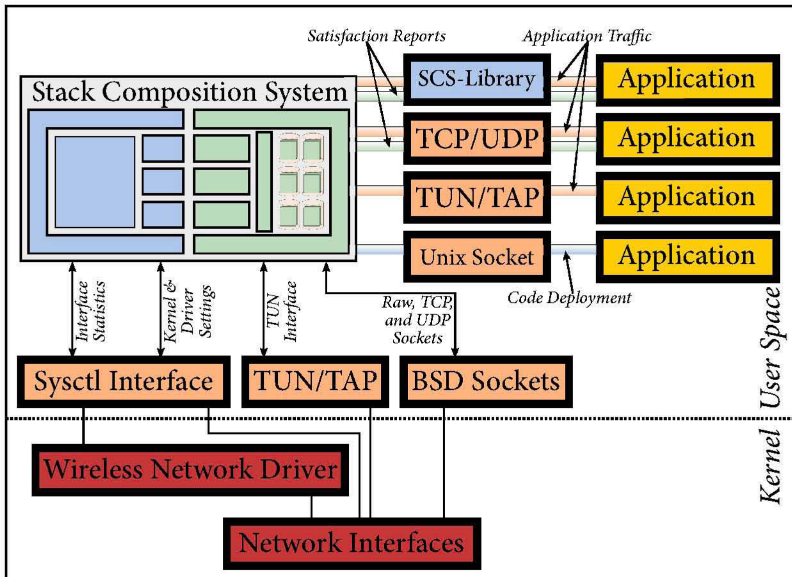


Figure B.1: *The interaction between the user-space stack composition system daemon and the operating system and local applications. We use BSD sockets and the TUN/TAP interface for communications, and extract network statistics, poll, and configure the operating system by means of the sysctls.*

To applications the composition system's data transfer interface is accessible either via a virtual network interface,[202] through BSD sockets,[303] or via a statically-linkable library that we provide. The aforementioned library provides additional features not offered by the standard communication facilities, e.g. the possibility to send satisfaction reports to the stack composition system as described in *Section 5.3.4*, or to specify detailed QoS requirements for the traffic on a per-flow basis. Access to the facilities for code deployment, which we described in *Section 3.7*, is provided through Unix sockets, and thus subject to the operating system-imposed access controls.

The stack steering system links the stack to virtual physical interfaces and application modules, which abstract the access to the operating system-specific communication facilities and thus enables us to implement stack modules in a more-or-less system-independent way. These modules and the stack steering system interact with each other by means of connectors that are assigned the physical or application service identifier, respectively.

The stack steering system forwards all incoming application data to a

module instance that offers the application service. The source and destination identifiers are derived from the incoming application data. If, for example, an UDP packet is received on the TUN/TAP-Interface, the source and destination identifiers consists of the corresponding IPv4 address, the IPv4 protocol field, and the UDP port. For incoming packets on the system's UDP or TCP ports, a run-time-configured destination is assigned which depends on the respective port. For outgoing packets, the appropriate data packet is re-assembled using the aforementioned identifiers.

The physical service provided by the stack steering system interacts with the operating system's network interfaces in a similar fashion. For outgoing packets over raw sockets, it encodes the caller's protocol identifier in the appropriate fields of the underlying protocol's header, e.g. the EtherType field for IEEE 802.3. Incoming packets are forwarded upwards into the stack according to the value encoded in the protocol field, in accordance with the principle that the sender defines the composition, as introduced in *Section 3.7.5*. When sending over an UDP socket provided by the host's operating system, a short header is prepended to each outgoing packet which consists of the source and destination addresses, the caller's protocol identifier and the length of the packet.

While the code-base itself is written with compatibility in mind and should compile on all POSIX-compliant systems, some system-dependent aspects were unavoidable. In particular the access to network-related measurement data depends on proprietary operating system interfaces, in some cases even necessitates access to the network interface drivers, and thus requires adaptation to the host system. We so far implemented and tested the necessary functionality for Mac OS X, FreeBSD, and Linux.

B.1.2 Simulation

For our simulations we utilise ns-3, a discrete-event driven open-source network simulator which is widely used for network research. It offers the possibility to specify both the experimentation scenarios – network topology, scheduled traffic, etc. – and the network stack architecture in C++ or Python. This enabled us to easily utilise the same code base for the simulation and standalone user-space implementation described above. Thanks to its easily extensible, modular, and object-oriented design, we were able to easily separate the operating system- and simulator-dependent aspects of the design from the common machinery, and only needed to implement a limited amount of "glue-code" to provide the interface to ns-3. For setting up the simulation, we implemented a helper application that, based on command line parameters, initialises and configures the stack composition sys-

tem, generates the network topology, and schedules application traffic. This application thus enabled us to easily script runs of experiments for various settings at once, and at higher speeds than in reality, enabling us to explore network configurations, traffic scenarios in far greater amount than possible in reality.

B.1.3 Realistic Physical Environment

As we mentioned before, one of our main arguments for the need for autonomous network stack adaptation is the inherent unpredictability of real networks. We therefore deployed our stand-alone implementation of the stack composition system on a physical test-bed, which enabled us to test the performance of the system over the real, unfiltered Internet and for realistic traffic loads. We spread our test bed across two sites. One part of the test-bed is hosted by the Computer Science department of the University of Basel, Switzerland, the other part is located roughly 70 km away in a residential area of Freiburg, Germany. In Basel we set up one server-class machine which is connected via 100M bit/s Ethernet to the university's backbone link and locally hosts an instance of the stack composition system. In Freiburg we installed two Soekris NET6501 [324] embedded network devices, one connected via Ethernet, and another one which provides both WLAN and Ethernet, which both also host one instance of the stack composition system. These machines are connected to the Internet via a very busy home-user class VDSL link. In addition to the experimental traffic, this link carries normal user traffic, and offers a total bandwidth of 50M bit/s down- and 10M bit/s up-stream. Behind each of the aforementioned machines we placed one laptop for the purpose of generating user traffic that is transmitted across the Internet via the stack composition system. This test-bed enables us to model various realistic scenarios as we can utilise home user-, embedded-, and server-class hardware, wired and wireless communications, reliable and fast Ethernet, as well as a somewhat more error-prone wireless connection. Furthermore, we can simulate various complementary network conditions by means of dummynet.

B.2 Implemented Protocol and Service Facilities

Since we intend to probe the stack composition system's ability to operate successfully in a realistic setting, we need to provide a network stack that is composable and configurable: Without actual implementations of protocols to experiment with and to measure the performance of, we cannot make

sufficient assumptions about the stack's utility and thus the stack composition system's behaviour. And as we intend to test some aspects of the system through communication across the Internet, we need to stay sufficiently compliant to the IETF's standards. In this section we introduce some of the protocols and modules we implemented for this purpose and utilised for our experiments.

B.2.1 Internet Protocol Implementations

We implemented the following communication protocols using the Unified Communication Interface introduced in *Section 3.7.3*. Since our system allows this protocols to be linked together in ways not envisioned by their designers, we had to make some modifications and extend the protocol functionality. Our extended versions of TCP, UDP, DCCP, IPv4, and Ethernet comply with the respective standards whenever the underlay they are connected to is one of those originally intended, i.e. when the transport-layer protocols are stacked on top of IPv4, or when IPv4 is connected to the Ethernet module. For different compositions, however, their behaviour is changed such as to still be able to operate: For example, UDP includes the appropriate IP header fields in its checksum calculation only when it is situated on top of IPv4, but accommodates different underlays (about whose operation it has no intrinsic knowledge) by simply omitting the underlay's header fields. Thus UDP can, for example, operate directly on top of Ethernet, in which case it would get assigned a proprietary value in the `EtherType` field, whereas IPv4 is identified by `0800h` as according to the IEEE specification.

Transmission Control Protocol (TCP) Apart from the basic functionality outlined in RFC 793[289], we implemented several modifications and extensions, most of which are also supported by several or all of the major current operating system stack implementations:

- RFC 1323[171]: Window Scale Option, Round-Trip Time Measurement (RTTM), and Protect Against Wrapped Sequence Numbers (PAWS);
- RFC 5681[14]: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery algorithms;
- RFC 3782[115]: NewReno Fast Recovery algorithm;
- RFC 2988[275]: Computation of the retransmission timer;
- RFC 2018[233]: Selective Acknowledgement options (SACK);

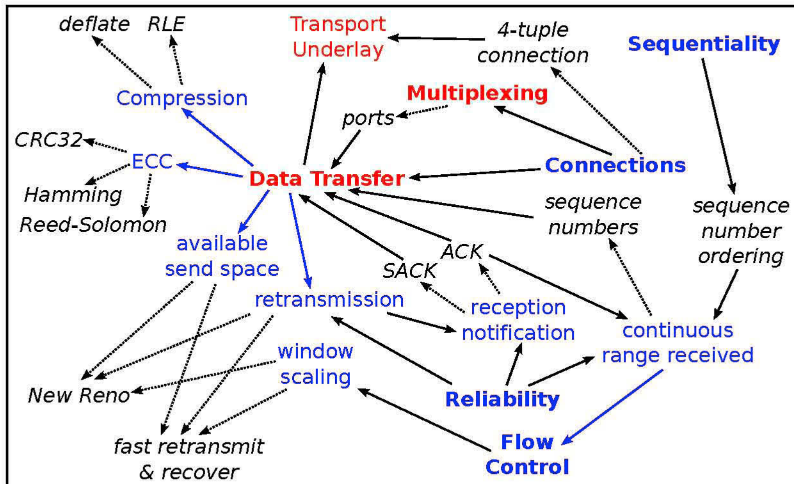


Figure B.2: *The modularization of TCP: The facilities defined RFC 793*[289] *are given in bold. Solid arrows denote dependencies, dotted ones implementations of a facility. Required facilities are red, optional (sub-)facilities and dependencies blue. Compression and error correction are our proprietary additions to the protocol.*

- RFC 1146[396]: Alternate Checksum options;
- RFC 3390[13]: Increasing the Initial Window size;
- RFC 3465[12]: Congestion Control with Appropriate Byte Counting (ABC).

As stated in RFC 793[289], TCP offers the following facilities: Basic Data Transfer, Reliability, Flow Control, Multiplexing, Connections, Precedence and Security. By modularising the design of TCP, i.e. splitting it into separate, but inter-dependent components, that can be enabled, disabled or replaced as needed, we gain the ability to tailor a transport protocol specific to the needs of the application. Disabling all facilities except for Data Transfer and Multiplexing, for example, would transform TCP into UDP[3]. The decomposition of functionalities and their interdependences is shown in *Figure B.2*. Only *Multiplexing* and *Data Transfer*, as well as the sub-modules these depend on, are fixed components in use within all compositions of our

[3] Functionality-wise, not w.r.t. to the actual packet encoding.

Modular TCP. All other facilities are optional and enabled through module control parameters. The actual functionality provided by the module, e.g. whether to use New Reno, is thus decided on by the evolution engine. Apart from the standard capabilities of TCP, we implemented extensions for error-correcting codes and compression, which we make available through proprietary TCP options.

User Datagram Protocol (UDP) Our implementation of UDP is fully compliant with RFC 768[286], and as such does not offer any stack composition system-configurable features, since the protocol specification does neither offer any features for configuration nor the possibility for extensions (as, for example, TCP does). Any standard-conforming implementation of UDP therefore is limited to providing just port-multiplexing and error-detection.

Datagram Congestion Control Protocol (DCCP) DCCP is a connection-oriented transport protocol that has been designed with the aim to address some of the presumed shortcomings of TCP and UDP. In particular, applications that do not require sequential or reliable transport, but need congestion control to guarantee TCP-friendliness (see e.g. RFC 5348[114]) are potential users of this protocol: Whereas it transmits data in the sequence it is received from the higher layers, the protocol itself does not guarantee sequentiality of or loss-free transmission, but does provide congestion control. Our implementation is fully compliant with RFC 4340[195], and implements the CCID2 and CCID3 congestion control methods as specified in RFC 4341[116], RFC 4342[117], and RFC 5348[114]. It offers controls to select the checksum coverage, congestion control method, and whether to use short sequence numbers.

Internet Protocol, Version 4 (IPv4) We provide a standard-conforming implementation of version 4 of the Internet Protocol, as defined in RFC 791[288]. We decided not to extend IP by means of proprietary options, since we planned to perform experiments over the Internet where – according to a study performed in 2005[120] – packets containing IP options are likely to be silently dropped.

B.2.2 Other Protocols

In addition to the communication protocols describe above, we developed the following protocols and modules for our system.

Payload Encoder Protocol (Codec) In addition to our proprietary extensions to TCP for the same purpose, we implemented a stand-alone protocol which offers error-correction and compression functionality. Since compression algorithms require access to the full and ordered encoded data, the missing guarantee for reliable and sequential data transmission here limits the compression functionality to individual packets, which can result in a lower compression ratio compared to the stream-based compression functionality of our TCP extensions. But in turn this protocol module can be placed at any arbitrary position in the stack, e.g. directly on top of Ethernet or IPv4, and thus provide compression and error correction facilities to arbitrary transport protocols, not just TCP.

Generic Communication Protocol (GCP) For practically deployable autonomous stack re-configuration, a transport abstraction is necessary which frees uses applications from the need to understand the intrinsic of the utilised stack. Application processes on the current Internet still require a rather deep understanding of the mechanics and structure of the Internet for communication: A web browser, for example, needs to resolve domain names into IP addresses, and understand that protocol identifiers such as http:// require a TCP connection to default port 80 on the remote host. To alleviate this problem, we provide applications with an underlay-independent way of accessing resources on the Internet, in a similar manner as URIs in theory should allow. GCP is a simple distributed registration, resolution, and forwarding protocol, in which applications access resources by means of an identifier, which the protocol resolves into a locator by means of a DHT that is shared by all instances of the stack composition system. This locator includes the protocols and addresses needed to access the resource across the network, for example, as TCP(80),IPv4(192.0.2.1);UDP(1111),ETH(FF:EE:DD:CC:BB:AA), which denotes a resource reachable via TCP port 80 on IPv4 address 192.0.2.1, as well as via UDP directly over Ethernet on the local network.

Compression Our compression modules offer the following algorithms for (de-)compressing data, which they expose through a common interface:

RLE compresses data by encoding continuous sections of identical bytes as two bytes, the length and the repeated byte's value. Incompressible blocks of bytes are usually denoted by the negative length of the block followed by the data. This method is commonly employed in graphic file formats such as TGA, ILBM, and PCX.

DEFLATE operates by combining Huffman coding with LZ77 compression and is standardised in RFC 1951 [78]. DEFLATE is one of the most commonly employed data compression algorithms and used e.g. in PNG, GZ, and ZIP files.

Error Correction We provide modules for the following error correction mechanisms, which are exposed through a common interface:

Cyclic Redundancy Check is an error-detection method that utilises the remainder of a polynomial division of the input data to detect changes. While relatively short and very fast to calculate, these methods can only detect errors, but not correct them. We provide several variants of 8 to 64-bit, e.g. those specified in RFC 1146 [396].

Hamming Codes are linear error-correcting codes that utilise parity bits to detect and correct errors in the encoding of a symbol. We provide the common $(8, 4)$-variant, which protects 8-bits of data with 4 parity bits, and thus is able to detect up to two, and correct one erroneous bit

Reed-Solomon Codes are cyclic error-correcting codes that can detect and correct up to $\lfloor \frac{k}{2} \rfloor$ faulty symbols (here: bits) through the addition of k additional symbols. Our implementation allows the selection of the number of symbols and the block size either by means of a control or directly through the API.

Distributed Registry At many places within its architecture the Internet is statically configured. The protocol numbers used by many transport protocols for (de-)multiplexing data packets, for example, are assigned by IANA. While protocols DNS or ARP provide a facility to translate higher-layer identifiers – e.g. domain names or IP addresses – into their lower-layer counterparts – (e.g. IP or Ethernet addresses – and vice-versa, these facilities are insufficient for our purposes: Since we wish to grant nodes the flexibility to independently evolve their stack configurations as well as to add new protocols, we need to provide functionality to register mappings between protocol identifiers and numbers, or between higher-layer identifiers and lower-layer addresses *and* protocol identifiers. For this purpose we provide a distributed registry mechanism, in which such mappings are registered and the looked up by nodes running the stack composition system. Our implementations of DNS and ARP, as well as our proprietary GCP, use the distributed registry, as do all parts of the system that need to (de-)multiplex between protocols.

Glossary

ns-3 Network Simulator, version 3, www.nsnam.org. 143, 146, 156, 157, 167, 199, 201

3GPP Standardization Body for mobile communications, originally named "3rd Generation Partnership Project", but now working on 4th generation networks. 214

administrator The physical entity responsibly for deploying and maintaining stack composition system. 131, 136–139

ADSL Asymmetric Digital Subscriber Line, ITU G.992. 4

All-IP In entirety based on the Internet Protocol, in particular IPv6. 2

ANN Artifical Neural Network. 20, 21, 23, 24, 28, 116, 189

API Application Programming Interface. 36, 66, 67, 199, 207, 217

application environment encompasses not only the running application and their needs, but also the conditions, state and setup of the network, as well as all other on-going traffic. 70, 215

ARP Address Resolution Protocol, RFC 826 [284]. 13, 36, 72–74, 207

BER Byte error rate, the probability of corruption, individually applied to every byte in a data packet.. 156

brute-force An algorithm or technique which operates by exhaustively enumerating all possible candidate solution and checking whether they meet the goal condition(s).. 104, 105

C++ An object-oriented, statically typed, free-form, multi-paradigm, programming language, see http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.. 29, 67, 201

callee The module instance which is invoked by a connector, similar to an object a method of which is executed in an object-oriented programming language. 73

caller The module instance which invokes a connector, similar to a calling a method of an object in an object-oriented programming language. 73

composer The stack composer provides the stack composition mechanics, i.e. it constructs and initialises the stack as outlined by a stack blueprint. This component of the stack composition system is introduced in *Section 3.5.2*.. 53, 57, 61–63, 66

Composition Tree A search and optimisation method which we designed specifically for the purpose of autonomous stack evolution process encompassed by the evolution engine and which is introduced in *Section 4.1.*. 6, 97, 100, 107–109, 147, 148, 150, 151, 154, 157–159, 161, 162, 164–171, 175, 182, 186, 188

compression Data compression operates by reducing the amount of statistical redundancy. In the context of the stack composition system, compression refers to lossless data compression.. 205

condition The condition or state of the network and the ongoing traffic, a synonym for situation.. 45, 46

configuration space The search space onto which the evolution logic is applied, i.e. the domain of all possible stack configurations.. 83, 84, 104, 106, 107, 171, 212

connector A strongly-typed reference to a module instance. Connectors can be likened to smart pointers to objects in e.g. C++. 59, 66, 67, 116, 148, 200

control Configuration parameters which influence the operation of a module, as described in *Section 3.4.2*. Continuous controls refer to parameter ranges in which the distance between values corresponds to larger changes in behaviour, whereas no such relation exists for nominal controls.. 67, 92, 148, 149, 160, 205

CPU Central Processing Unit. 175

DCCP Datagram Congestion Control Protocol, RFC 4340[195]. 63, 118, 148, 150, 157, 163, 164, 174, 203, 205

decision module A module class whose purpose is to control and direct the execution and packet processing flow through the stack in conjunction with in-stack redirectors introduced in *Section 4.3.1.*. 56, 117, 118

DEFLATE A data compression algorithm, RFC 1951[78].. 158, 174, 176, 177, 207

DHT Distributed Hash Table, see e.g. Chord[333].. 36, 206

DNS Domain Name System, RFC 1035[247]. 36, 71, 207

DoS Denial-of-Service, the attempt to make a network service or resource unavailable for its intended purpose.. 128, 134, 137

DTN Delay-Tolerant Networking, also Disruption-Tolerant Networking, RFC 4838[49],5050. 193

dummynet A FreeBSD system for emulating the effects of bandwidth limitations, propagation delays, bounded-size queues, and packet losses, `http://info.iet.unipi.it/~luigi/dummynet/`. 202

EBNF Extended Backus-Naur Form, a meta-syntax notation for expressing context-free grammars.. 57, 110, 138, 140

ECN Explicit Congestion Notification, RFC 3168[294]; sadly not widely available in the current Internet. 16

environment The state and configuration of the network and traffic, including all aspects which are out of the control of the current node, but influence its operation.. 45, 124

error-correcting codes See forward error correction.. 155, 156, 158, 205, 212

Ethernet Ethernet, IEEE 802. 67, 69, 71–74, 157, 173, 174, 176–178, 202, 203, 206, 214

evolution engine The part of the stack composition system which implements the evolution logic.. 44, 52–54, 57, 59–62, 64, 65, 72, 79–81, 121, 144, 145, 148, 150, 155, 170, 178, 182, 187, 191

evolution logic The algorithms that manifest long-term evolution, and which we introduce in *Section 4.1.*. 48, 49, 52, 60, 65, 67, 68, 77, 80–84, 104, 106–108, 116–119, 129, 135, 139, 145, 148–150, 153, 155, 157, 159, 161, 163, 165, 167, 170–172, 175, 210, 211

evolution quality In the context of stack evolution we define quality as the fitness achieved by the best stack derived from any stack blueprint in a particular generation.. 83, 108, 148, 153, 163, 165, 178

evolution speed In the context of stack evolution we define speed as the time measured in generations – needed until the fitness of the best stack exceeds a specific threshold with sufficiently high probability.. 83, 102, 108, 148, 165, 171

Evolutionary Algorithm Evolutionary Algorithm, a class of machine learning method which includes e.g. Genetic Algorithm, and which we introduce in *Section 2.6.1*.. 22, 81, 87, 93, 97, 99, 106–109, 129, 148, 150, 153, 155, 157, 159, 160, 162, 166, 168–171, 186, 188, 212

experiment run We use the word experiment run to indicate a specific and distinct experiment. For example, two executions of the simulator with different random seeds constitute two distinct runs, while using the same seed would indicate to instances of the same run. Performing two experiments on the real-world set up always represent two distinct runs, as the conditions are assumed to always be distinct.. 145, 153, 154, 176, 177

fitness A scalar measure of utility which defines the proximity of a candidate solution to the objective or goal state.. 6, 48, 52, 58, 70, 79, 81, 83, 121, 132, 136, 138, 143, 187, 191, 212

fitness function An objective or utility function which calculates the fitness of a candidate solution.. 4, 20, 49, 53, 54, 57, 64, 84, 106, 122, 128, 131, 132, 135, 137–140, 144, 163, 212

fitness landscape The distribution of the fitness with regard to the configuration space. The fitness landscape is the codomain of fitness function $F(x)$ evaluated for all possible stack configurations.. 82–84, 102, 106, 171, 190

forward error correction see error-correcting codes. 57, 60, 118, 155, 160

FPGA Field-Programmable Gate Array. 32

FreeBSD `www.freebsd.org`. 31, 201, 211, 217

FTP File Transfer Protocol, RFC 959 [290]. 2, 74, 76, 139

fuzzing A testing methodology which provides invalid, unexpected, or random data as input to the entity that is to be tested.. 195

GCP Generic Communication Protocol, a protocol which abstracts application-level access to the stack functionality, which we introduce in *Section B.2.2*.. 148, 157, 160, 161, 206, 207

generation Generation denotes a set of stack blueprints of stack configurations derived by an Evolutionary Algorithm. These algorithms generate a new set of stack blueprints based on information about the fitness and layout of the previous generations similar to biological procreation. Thus the terms parent, offspring or children are also often used to refer to the previous or successive generation.. 52, 54, 64, 65, 81, 83, 84, 129, 145, 176, 211

Genetic Algorithm Genetic Algorithm, a machine learning method introduced in *Section 2.6.1*.. 87, 108, 119, 152, 160, 211

hill-climbing A local search technique applicable to optimisation problems [308].. 102

HTML Hyper-Text Markup Language, `http://www.w3.org/standards/webdesign/htmlcss`.. 156, 174

HTTP Hypertext Transfer Protocol, defined in RFC 2616 [112].. 3, 21, 67

I/O input/output. 29

IANA Internet Assigned Numbers Authority, `iana.org`.. 36, 207

ICMP Internet Control Message Protocol, RFC 792 [287]. 76

IEEE Institute of Electrical and Electronics Engineers, `www.ieee.org`. 43, 156, 203, 214

IETF Internet Engineering Task Force, `www.ietf.org`. 157, 203

in-stack redirector Modules placed into the stack which implement branching between independent sub-stacks at the packet- or flow-level, as introduced in *Section 4.3.*. 52, 53, 56, 57, 62, 64, 65, 75, 80, 116, 118, 153

instance An initialized module, which is created and set up by the stack composer in accordance to a stack blueprint. 116, 216

interface The point of interaction between unrelated objects, in object-oriented programming languages the specification of methods and values for cooperation.. 117

IP address routable node address, 32-bit for IPv4, 128-bit for IPv6. 2, 73

IPC Inter-Process Communication. 69, 137

IPng IPng, RFC 1726 [271]. 3

IPv4 Internet Protocol version 4, RFC 791 [288]. 2, 36, 67, 69, 71–74, 76, 146, 148, 150, 156, 157, 162, 174, 194, 203, 206, 213

IPv6 Internet Protocol version 6, RFC 2460 [75]. 2, 71, 194, 209, 213

k-means A method of cluster analysis [150].. 21, 110, 111, 114, 115, 172, 189

Linux Another Unix-like operating system kernel, often deployed together with the GNU application base.. 199, 201, 217

LTE 3GPP Long-Term Evolution, the 4-th generation of mobile telephony networks. 2, 3

Mac OS X `developer.apple.com`. 201, 217

mDNS Multicast DNS, see RFC 6762 **?** .. 73

MDP Markov Decision Process, see e.g. Sutton [343].. 24, 25

meta-protocol Our abstraction of communication protocols, which incorporates the address resolution as well as packet forwarding, etc.. 72

minimum spanning tree A spanning tree with weight less than or equal to the weight of every other spanning tree. A spanning tree is connected, undirected, and weighted graph that connects all vertices together [81].. 99

module A subset of protocol or service functionality within the stack composition system, which is implemented a monolothic entity, a.k.a. micro-protocol or service in SILO [96]. The stack is composed of modules, which we introduce in *Section 3.4.1.*. 73, 76, 116, 122

MTU Maximum Transfer Unit, is the size of the largest payload that a protocol layer, e.g. Ethernet, is able to forward.. 156

NAT Network Address Translation, RFC 3022 [329]. 2

NTP Network Time Protocol, RFC 5905 [242]. 129

operating system The software layer which abstracts and manages access to the system hardware and provides services to the programs running on top of it.. 6, 32, 33, 35, 62, 66, 122, 150, 173, 182, 187, 195, 199–201, 203, 214

opportunistic networking A network architecture, such as HAGGLE [339], which aims to enable communication when network connectivity is intermittent.. 193

packet-switched network A networkign paradigm which groups all exchanged data into chunks or blocks, which are called packets. The opposite of circuit-switched networking, which allocates dedicated commmunication channels for each connection.. 2

PAWS TCP Extension PAWS, "Protect Against Wrapped Sequence Numbers", RFC 1323 [171]. 195

peer-to-peer A distributed application architecture that partitions tasks or work loads between peers, often based on abstract overlay networks.. 3, 4, 23, 25, 26, 30, 31

population The group of all stack configuration stack blueprints and associated sensor measurements evolved under a specific situation, see introductio of the mid-term adaptation logic in *Section 4.2*.. 50, 52–54, 64, 65, 79, 80, 86, 110, 175, 176, 178

population selector The components of the stack composition system which manages access to the populations based on the output of the situational classifier.. 53, 54, 64, 65, 118, 172, 175, 178

POSIX Portable Operating System Interface, see e.g. IEEE Std 1003.1-2008, or `https://collaboration.opengroup.org/external/pasc.org/plato/`. 71, 201

Python A versatile scripting language, in widespread use e.g. for rapid prototyping, `www.python.org`. 201

QoS Quality of Service. 19, 20, 23, 24, 31, 74, 188, 200

random probing The random probing algorithm operates by repeatedly generating random stack configurations and is introduced in *Section 4.1.3*. 85, 93, 105–107, 151, 152, 157, 160–162, 168–170

Rapidly-Exploring Random Tree A search method and data structure for searching high-dimensional non-convex spaces [209].. 94–97, 99, 100, 107, 188

RLE Run-Length Encoding, a simple data compression scheme. 158, 174, 206

roulette-wheel selection A fitness-proportionate selection method which selects candidate solutions with a probability proportionate to their associated fitness.. 87, 99

RTT Round-Trip-Time, the delay between the time when a message is sent and the time when the acknowledgement for this message is received.. 123, 197

satisfaction The measure of complacency of a node's stack composition system. Higher satisfaction results in less exploration of new stack configurations, i.e. less experimentation, which in turn implies a more stable modus operandi.. 136, 137, 139, 140, 187, 190, 200

segmentation fault An error signal generated when the CPU tries to access an inaccessible memory location.. 135

sensor Our facilities for abstracting access to measurement and state data, as introduced in *Section 5.1*.. 57, 62, 118, 122, 137, 138, 140

service A discrete set of software or protocol functionality, see also module.. 68, 117, 198

sidechannel A reliable means of communications which uses a known-good stack configuration instead of the trialled stack configuration to guarantee a high probability of delivery. This know-good stack can be either a common stack variant (e.g. the TCP/IP-Stack) or a stack which proved reliable when tested in multiple application environment.. 123, 134, 136, 137

situation The conditionas and state of the network, traffic, and applications, i.e. everything that can have an influence on the performance of the system.. 45, 47, 49, 50, 53, 54, 64, 65, 79, 82, 84, 110, 119, 121, 130, 133, 135–138, 175, 176, 178, 190, 191, 210

situational classifier The component of the stack composition system which is responsible for grouping situations into different classes based on the similarity between them according to preselected criteria and which is introduced in *Section 4.2*.. 52–54, 57, 64, 65, 80, 110, 118, 172

SNMP Simple Network Management Protocol, RFC 3411 [155]-3418. 13, 34, 123

stack The protocol stack is a directed graph composed of (micro-)protocol modules, which specifies the possible paths and interactions of protocols and communication services. Please refer to *Section 3.7.5* for a discussion of how our design regulates the control flow between these modules.. 45, 54, 121, 134, 176, 216

stack blueprint A "plan" which defines how many instances of each module class are present in a stack, how they are connected with each other, and how they are configured.. 52, 54, 57, 61–65, 79–83, 110, 129, 157, 167, 211, 212

stack composition system Our framework for stack evolution.. 7, 45, 47, 48, 50, 52–55, 57, 59, 62, 63, 66, 70, 71, 75–79, 84, 106, 115–119, 121, 123, 124, 127–133, 135–139, 144–150, 152, 154, 155, 157, 158, 163, 170, 174, 178, 182, 183, 185, 186, 189, 193–203, 205–207, 209–211, 217

stack configuration The composition of a stack out of individual service modules, as well as the configuration of these module instances.. 4–7, 16, 18, 20, 30, 32, 45, 47–52, 60, 64–67, 70, 75, 79–82, 84, 86, 87, 94, 95, 104, 109, 113–115, 121, 124, 126, 128–131, 133–136, 145, 148–150, 152, 153, 157, 160–163, 166, 170, 171, 175–179, 182, 186, 188, 190, 195, 210, 212, 215

stack steering system The part of the stack composition system which links the stack to outside and which controls the stack operations as well as the adaptation process, as described in *Section 3.5.1*.. 53, 54, 59, 62, 65, 86, 122, 129, 130, 134, 135, 199–201

sub-trial A sub-trial is an independently scheduled experimentation phase, during which one specific stack configuration is tested. See also trial.. 122, 130–133, 135, 138, 148, 150, 153, 154, 157, 167, 171, 175, 178

TCP Transmission Control Protocol, RFC 793 [289]. 3, 14–16, 19, 22, 28, 31, 32, 34, 36, 50, 51, 63, 67, 69–71, 73, 74, 115, 118, 123, 128, 146, 148–150, 153, 157, 174, 193–196, 203, 205, 206

TCP Friendly Rate Control TCP Friendly Rate Control, RFC 5348 [114]. 196

TCP-friendliness Congestion control mechanisms designed to operate together with concurrent TCP traffic, such that the achieved throughput is relatively fairly distributed between the flows, see TCP Friendly Rate Control.. 34, 69, 194, 196

trial A trial is the sum of all sub-trials performed for a particular stack configuration. Due to the noisy nature of experimentation, one single sample, i.e. experiment, is not enough, thus the stack composition system performs multiple sub-trials and averages the results, as described in *Section 5.3.2*.. 130, 148

trust The trust relationship between nodes on which the stack composition system is deployed is described in *Section 5.3.4*.. 134, 136

UDP User Datagram Protocol, RFC 768 [286].. 3, 16, 31, 63, 67, 69, 115, 118, 148, 149, 157, 162–164, 174, 177, 190, 194, 196, 203, 205

Unified Communication Interface The API utilised by all communication meta-protocols provided by the stack composition system.. 69, 71, 150, 203

Unix Originally a multi-user and multi-tasking operating system developed at Bell Labs, but now commonly used to refers to all Unix-like or UN*X operating systems, such as FreeBSD, Mac OS X, Linux, etc.. 29, 30

URI Uniform Resource Identifier, RFC 3968 [45].. 206

VDSL Very-high-bit-rate digital subscriber line.. 125, 131, 156, 174, 202

VLAN Virtual Local Area Network. 3

VoIP Voice-over-IP. 2, 74, 126, 139, 140

VPN Virtual Private Network. 3

Wi-Fi Wireless networking technology, IEEE 802.11. 3

WiMAX Worldwide Interoperability for Microwave Access. 3

WLAN Wireless Local Area Network. 156, 173, 174, 176–178, 202

Bibliography

1 The NS-3 network simulator. http://www.nsnam.org.

2 3GPP IP Multimedia Subsystem. http://www.3gpp.org/Technologies/Keywords-Acronyms/article/ims/.

3 The FP7 4WARD Project. http://www.4ward-project.eu.

4 Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, Jr., T.F., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R. Amorphous computing. Commun. ACM, 43 (5):pp. 74–82, 2000.

5 Agarwal, M., Bhat, V., Liu, H., Matossian, V., Putty, V., Schmidt, C., Zhang, G., Zhen, L., Parashar, M., Khargharia, B., Hariri, S. AutoMate: enabling autonomic applications on the grid. In Autonomic Computing Workshop. 2003. Proceedings of the, pp. 48–57 (June).

6 Ahn, C.W., Ramakrishna, R.S. A genetic algorithm for shortest path routing problem and the sizing of populations. Evolutionary Computation, IEEE Transactions on, 6 (6):pp. 566–579, 2002.

7 AKARI. http://akari-project.nict.go.jp.

8 Akyildiz, I.F., Lee, W.Y., Vuran, M.C., Mohanty, S. NeXt generation/dynamic spectrum access/cognitive radio wireless networks: A survey. Computer Networks, 50 (13):pp. 2127 – 2159, 2006.

9 Al-Shraideh, F. Host Identity Protocol. In Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on, pp. 203–203 (2006).

10 Alagar, V.S., Alagar, V.S., Achuthan, R., Haydar, M., Muthiayen, D., Ormandjieva, O., Zheng, M. A rigorous approach for constructing self-evolving real-time reactive systems. Information & Software Technology, 45 (11):pp. 743–761, 2003.

11 Ali-Yahiya, T., Bullot, T., Beylot, A.L., Pujolle, G. A Cross-Layer Based Autonomic Architecture for Mobility and QoS Supports in 4G Networks. In Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, pp. 79–83 (2008).

12 Allman, M. TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465 (Experimental),

2003. URL http://www.ietf.org/rfc/rfc3465.txt.

13 Allman, M., Floyd, S., Partridge, C. Increasing TCP's Initial Window. RFC 3390 (Proposed Standard), 2002. URL http://www.ietf.org/rfc/rfc3390.txt.

14 Allman, M., Paxson, V., Blanton, E. TCP Congestion Control. RFC 5681 (Draft Standard), 2009. URL http://www.ietf.org/rfc/rfc5681.txt.

15 Ammar, M. Why we still don't know how to simulate networks. In Simulation Symposium, 2005. Proceedings. 38th Annual, p. 3, (IEEE2005).

16 Autonomic Network Architecture Project. http://www.ana-project.org.

17 Anagnostakis, K.G., Ioannidis, S., Miltchev, S., Greenwald, M., Smith, J.M., Ioannidis, J. Efficient packet monitoring for network management. In Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP, pp. 423–436, (IEEE2002).

18 Anderson, T., Peterson, L., Shenker, S., Turner, J. Overcoming the Internet impasse through virtualization. Computer, 38 (4):pp. 34–41, 2005.

19 Ashby, W.R. Principles of the self-organizing system. Principles of Self-organization, pp. 255–278, 1962.

20 Atzori, L., Iera, A., Morabito, G. The internet of things: A survey. Computer Networks, 54 (15):pp. 2787–2805, 2010.

21 Babaoglu, O. The Self-star Vision. Self-star Properties in Complex Information Systems, pp. 397–397, 2005.

22 Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T. Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst., 1 (1):pp. 26–66, 2006.

23 Babaoglu, O., Meling, H., Montresor, A. Anthill: a framework for the development of agent-based peer-to-peer systems. In Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on, pp. 15–22 (2002).

24 Bäck, T. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. Parallel problem solving from nature, 2:pp. 85–94, 1992.

25 —. Optimal Mutation Rates in Genetic Search. In Proceedings of the 5th International Conference on Genetic Algorithms, pp. 2–8, (Morgan Kaufmann Publishers Inc.1993).

26 —. Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms, (Oxford University Press on Demand1996).

27 Balakrishnan, H., Lakshminarayanan, K., Ratnasamy, S., Shenker, S., Stoica, I., Walfish, M. A Layered Naming Architecture for the Internet. In Proc. ACM SIGCOMM, (Portland, OR2004).

28 Balamuralidhar, P., Prasad, R. A Context Driven Architecture for Cognitive Radio Nodes. Wireless Personal Communications, 45:pp. 423–434, 2008.

29 Barolli, L., Koyama, A., Shiratori, N. A QoS routing method for ad-hoc networks based on genetic algorithm. In Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on, pp. 175–179 (2003).

30 Bartz-Beielstein, T., Lasarczyk, C.W., Preuß, M. Sequential parameter optimization. In Evolutionary Computation, 2005. The 2005 IEEE Congress on, vol. 1, pp. 773–780, (IEEE2005).

31 Bellardo, J., Savage, S. Measuring packet reordering. In Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, IMW '02, pp. 97–105, (ACM, New York, NY, USA2002).

32 Bennani, M., Menasce, D. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. In Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on, pp. 229–240 (2005).

33 Bertsekas, D.P., Gallager, R.G., Humblet, P. Data networks, vol. 2, (Prentice-Hall International1992).

34 Bhatti, N.T., Schlichting, R.D. A system for constructing configurable high-level protocols. SIGCOMM Comput. Commun. Rev., 25 (4):pp. 138–150, 1995.

35 Bicket, J.C. Bit-rate selection in wireless networks, 2005.

36 BIONETS. http://www.bionets.eu.

37 Blumenthal, M.S., Clark, D.D. Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world. ACM Trans. Internet Technol., 1 (1):pp. 70–109, 2001.

38 Bouabene, G., Jelger, C., Tschudin, C., Schmid, S., Keller, A., May, M. The autonomic network architecture (ANA). Selected Areas in Communications, IEEE Journal on, 28 (1):pp. 4–14, 2010.

39 Braden, R. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), 1989. URL http://www.ietf.org/rfc/rfc1122.txt. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633.

40 Braden, R., Faber, T., Handley, M. From protocol stack to protocol heap: role-based architecture. SIGCOMM Comput. Commun. Rev., 33 (1):pp. 17–22, 2003.

41 Bridges, P., Wong, G., Hiltunen, M., Schlichting, R., Barrick, M. A Configurable and Extensible Transport Protocol. Networking, IEEE/ACM Transactions on, 15 (6):pp. 1254–1265, 2007.

42 Brooks, R.A. Intelligence without reason. In COMPUTERS AND THOUGHT, IJCAI-91, pp. 569–595, (Morgan Kaufmann1991).

43 Bullot, T., Gaïti, D., Pujolle, G., Zimmermann, H. A piloting plane for controlling wireless devices. Telecommunication Systems, 39:pp. 195–203, 2008.

44 Busoniu, L., Babuska, R., De Schutter, B. A comprehensive survey of multiagent reinforcement learning. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 38 (2):pp. 156–172, 2008.

45 Camarillo, G. The Internet Assigned Number Authority (IANA) Header Field Parameter Registry for the Session Initiation Protocol (SIP). RFC 3968 (Best Current Practice), 2004. URL http://www.ietf.org/rfc/rfc3968.txt.

46 Candea, G., Brown, A., Fox, A., Patterson, D. Recovery-oriented computing: building multitier dependability. Computer, 37 (11):pp. 60–67, 2004.

47 Candea, G., Kiciman, E., Kawamoto, S., Fox, A. Autonomous recovery in componentized Internet applications. Cluster Computing, 9:pp. 175–190, 2006.

48 Cantin, F., Goebel, V., Gueye, B., Kaafar, D., Leduc, G., Siekkinen, M., Xiao, J., Young, M. ANA Deliverable D3.8 – Self-Optimization Mechanisms. Deliverable, ANA Project, 2008.

49 Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., Weiss, H. Delay-Tolerant Networking Architecture. RFC 4838 (Informational), 2007. URL http://www.ietf.org/rfc/rfc4838.txt.

50 Chandranmenon, G.P., Varghese, G. Trading packet headers for packet processing. IEEE/ACM Trans. Netw., 4 (2):pp. 141–152, 1996.

51 Cheshire, S., Aboba, B., Guttman, E. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), 2005. URL http://www.ietf.org/rfc/rfc3927.txt.

52 Chess, D.M., Segal, A., Whalley, I., White, S.R. Unity: Experiences with a Prototype Autonomic Computing System. Autonomic Computing, International Conference on, 0:pp. 140–147, 2004.

53 Chiang, M., Low, S., Calderbank, A., Doyle, J. Layering as Optimization Decomposition: A Mathematical Theory of Network Architectures. Proceedings of the IEEE, 95 (1):pp. 255–312, 2007.

54 Chien, C., Srivastava, M., Jain, R., Lettieri, P., Aggarwal, V., Sternowski, R. Adaptive radio for multimedia wireless links. Selected Areas in Communications, IEEE Journal on, 17 (5):pp. 793–813, 1993.

55 Cho, K., et al. Managing traffic with ALTQ. In Proceedings of USENIX, 1999 Annual Technical Conference: FREENIX Track, Monterey CA, pp. 121–128 (1999).

56 ChoiceNet. https://code.renci.org/gf/project/choicenet/.

57 Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M. Planetlab: an overlay testbed for broad-coverage services. ACM SIGCOMM Computer Communication Review, 33 (3):pp. 3–12, 2003.

58 Claffy, K.C., Polyzos, G.C., Braun, H.W. Application of sampling methodologies to network traffic characterization. SIGCOMM Comput. Commun. Rev., 23 (4):pp. 194–203, 1993.

59 Claise, B. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), 2004. URL http://www.ietf.org/rfc/rfc3954.txt.

60 —. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), 2008. URL http://www.ietf.org/rfc/rfc5101.txt.

61 Clark, D. The design philosophy of the DARPA Internet protocols. In ACM SIGCOMM Computer Communication Review, vol. 18 (4), pp. 106–114, (ACM1988).

62 Clark, D., Braden, R., Sollins, K., Wroclawski, J., Katabi, D. New Arch: future generation internet architecture. Tech. rep., DTIC Document, 2004.

63 Clark, D.D., Partridge, C., Ramming, J.C., Wroclawski, J.T. A knowledge plane for the internet. In SIGCOMM ’03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 3–10, (ACM, New York, NY, USA2003).

64 Clark, D.D., Sollins, K., Wroclawski, J., Faber, T. Addressing reality: an architectural response to real-world demands on the evolving Internet. SIGCOMM Comput. Commun. Rev., 33 (4):pp. 247–257, 2003.

65 Clark, D.D., Tennenhouse, D.L. Architectural considerations for a new generation of protocols. In ACM SIGCOMM Computer Communication Review, vol. 20 (4), pp. 200–208, (ACM1990).

66 Clark, D.D., Wroclawski, J., Sollins, K.R., Braden, R. Tussle in cyberspace: Defining tomorrow’s Internet. In In Proc. ACM SIGCOMM, pp. 347–356 (2002).

67 Clean Slate. http://cleanslate.stanford.edu.

68 Collins, J., Luca, C. The effects of visual input on open-loop and closed-loop postural control mechanisms. Experimental Brain Research, 103:pp. 151–163, 1995.

69 Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, S.R.T. Finally, a use for componentized transport protocols. In HotNets IV, vol. 13 (2005).

70 Crane, S., Dulay, N. A configurable protocol architecture for CORBA environments. In Autonomous Decentralized

Systems, 1997. Proceedings. ISADS 97., Third International Symposium on, pp. 187–194 (1997).

71 Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V. Gigascope: A stream database for network applications. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 647–651, (ACM2003).

72 Crowcroft, J., Hand, S., Mortier, R., Roscoe, T., Warfield, A. Plutarch: an argument for network pluralism. ACM SIGCOMM Computer Communication Review, 33 (4):pp. 258–266, 2003.

73 Crowcroft, J., Oechslin, P. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. ACM SIGCOMM Computer Communication Review, 28 (3):pp. 53–69, 1998.

74 Decasper, D., Dittia, Z., Parulkar, G., Plattner, B. Router plugins: a software architecture for next generation routers. In Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication, SIGCOMM '98, pp. 229–240, (ACM, New York, NY, USA1998).

75 Deering, S., Hinden, R. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), 1998. URL http://www.ietf.org/rfc/rfc2460.txt. Updated by RFCs 5095, 5722, 5871, 6437, 6564.

76 Dellarocas, C., Klein, M., Shrobe, H. An architecture for constructing self-evolving software systems. In ISAW '98: Proceedings of the third international workshop on Software architecture, pp. 29–32, (ACM, New York, NY, USA1998).

77 Demestichas, P., Stavroulaki, V., Boscovic, D., Lee, A., Strassner, J. m@ANGEL: autonomic management platform for seamless cognitive connectivity to the mobile internet. Communications Magazine, IEEE, 44 (6):pp. 118–127, 2006.

78 Deutsch, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), 1996. URL http://www.ietf.org/rfc/rfc1951.txt.

79 Devroye, N., Vu, M., Tarokh, V. Cognitive radio networks. Signal Processing Magazine, IEEE, 25 (6):pp. 12–23, 2008.

80 Di Caro, G., Dorigo, M. AntNet: Distributed Stigmergetic Control for Communications Networks. Journal of Artificial Intelligence Research, 9:pp. 317–365, 1998.

81 Dijkstra, E.W. A note on two problems in connexion with graphs. Numerische mathematik, 1 (1):pp. 269–271, 1959.

82 Dobson, G., Sanchez-Macian, A. Towards Unified QoS/SLA Ontologies. In Services Computing Workshops, 2006. SCW '06. IEEE, pp. 169–174 (2006).

83 Dobson, S., Bailey, E., Knox, S., Shannon, R., Quigley, A. A first approach to the closed-form specification and analysis of an autonomic control system. In ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 229–237, (IEEE Computer Society, Washington, DC, USA2007).

84 Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F. A survey of autonomic communications. ACM Trans. Auton. Adapt. Syst., 1 (2):pp. 223–259, 2006.

85 Domingue, J., Fensel, D., González-Cabero, R. SOA4All, enabling the SOA revolution on a world wide scale. In Semantic Computing, 2008 IEEE International Conference on, pp. 530–537, (IEEE2008).

86 Dorigo, M., Di Caro, G. Ant colony optimization: a new meta-heuristic. In Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on, vol. 2, pp. –1477 Vol. 2 (1999).

87 Dorigo, M., Maniezzo, V., Colorni, A. Ant system: optimization by a colony of cooperating agents. Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on, 26 (1):pp. 29–41, 1996.

88 Dornbush, S., Joshi, A. StreetSmart Traffic: Discovering and Disseminating Automobile Congestion Using VANETs. In Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th, pp. 11–15 (2007).

89 Douglas, P.H. The Cobb-Douglas production function once again: its history, its testing, and some new empirical values. The Journal of Political Economy, pp. 903–915, 1976.

90 Dowling, J., Curran, E., Cunningham, R., Cahill, V. Using feedback in collaborative reinforcement learning to adaptively optimize MANET routing. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 35 (3):pp. 360–372, May.

91 Doyle, L., Sutton, P., Nolan, K., Lotze, J., Ozgul, B., Rondeau, T., Fahmy, S., Lahlou, H., DaSilva, L. Experiences from the Iris Testbed in Dynamic Spectrum Access and Cognitive Radio Experimentation. In New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on, pp. 1–8 (2010).

92 Dressler, F., Akan, O.B. A survey on bio-inspired networking. Computer Networks, 54 (6):pp. 881 – 900, 2010.

93 Duffield, N. Fair sampling across network flow measurements. SIGMETRICS Perform. Eval. Rev., 40 (1):pp. 367–378, 2012.

94 Duffield, N., Grossglauser, M. Trajectory sampling for direct traffic observation. Networking, IEEE/ACM Transactions on, 9 (3):pp. 280–292, 2001.

95 Duffield, N., Lund, C., Thorup, M. Learn more, sample less: control of volume and variance in network measurement. Information Theory, IEEE Transactions on, 51 (5):pp. 1756–1775, 2005.

96 Dutta, R., Rouskas, G., Baldine, I., Bragg, A., Stevenson, D. The SILO Architecture for Services Integration, controL, and Optimization for the Future Internet. In Communications, 2007. ICC '07. IEEE International Conference on, pp. 1899–1904 (2007).

97 E., S.I. A futures market in computer time. Communications of the ACM, 6 (11):pp. 449—451, 1968.

98 Eckhardt, D., Steenkiste, P. Improving wireless LAN performance via adaptive local error control. In Network Protocols, 1998. Proceedings. Sixth International Conference on, pp. 327–338 (1998).

99 Eiben, A. Evolutionary Computing and Autonomic Computing: Shared Problems, Shared Solutions? In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. Moorsel, M. Steen (eds.), Self-star Properties in Complex Information Systems, vol. 3460 of Lecture Notes in Computer Science, pp. 36–48, (Springer Berlin Heidelberg2005).

100 Eiben, A., Horvath, M., Kowalczyk, W., Schut, M. Reinforcement Learning for Online Control of Evolutionary Algorithms. In S. Brueckner, S. Hassas, M. Jelasity, D. Yamins (eds.), Engineering Self-Organising Systems, vol. 4335 of Lecture Notes in Computer Science, pp. 151–160, (Springer Berlin Heidelberg2007).

101 Eiben, A., Schut, M., Wilde, A. Is Self-adaptation of Selection Pressure and Population Size Possible? – A Case Study. In T. Runarsson, H.G. Beyer, E. Burke, J. Merelo-Guervós, L. Whitley, X. Yao (eds.), Parallel Problem Solving from Nature - PPSN IX, vol. 4193 of Lecture Notes in Computer Science, pp. 900–909, (Springer Berlin Heidelberg2006).

102 Eiben, A.E., Hinterding, R., Michalewicz, Z. Parameter control in evolutionary algorithms. Evolutionary Computation, IEEE Transactions on, 3 (2):pp. 124–141, 1999.

103 El Baz, D., Nguyen, T.T. A Self-adaptive Communication Protocol with Application to High Performance Peer to Peer Distributed Computing. In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, pp. 327–333 (2010).

104 Estan, C., Keys, K., Moore, D., Varghese, G. Building a better NetFlow. SIGCOMM Comput. Commun. Rev., 34 (4):pp. 245–256, 2004.

105 Estan, C., Varghese, G. New directions in traffic measurement and accounting. SIGCOMM Comput. Commun. Rev., 32 (4):pp. 323–336, 2002.

106 Exposito, E., Sénac, P., Garduno, D., Diaz, M., Urueña, M. Deploying new QoS aware transport services. In Protocols and Systems for Interactive Distributed Multimedia, pp. 141–153, (Springer2002).

107 Fabiunke, M. Parallel distributed constraint satisfaction. Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA-99), pp. 1585–1591, 1999.

108 Fall, K. A delay-tolerant network architecture for challenged internets. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 27–34, (ACM2003).

109 Feldmeier, D., McAuley, A., Smith, J., Bakin, D., Marcus, W., Raleigh, T. Protocol boosters. Selected Areas in Communications, IEEE Journal on, 16 (3):pp. 437–444, 1998.

110 Feng, X., Pechen, A., Jha, A., Wu, R., Rabitz, H. Global optimality of fitness landscapes in evolution. Chemical Science, 3 (3):pp. 900–906, 2012.

111 Fernandez-Prieto, J., Canada-Bago, J., Gadeo-Martos, M., Velasco, J.R. Optimisation of control parameters for genetic algorithms to test computer networks under realistic traffic loads. Applied Soft Computing, 11 (4):pp. 3744 – 3752, 2011.

112 Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFCs 2817, 5785, 6266, 6585.

113 Fiuczynski, M.E., Bershad, B.N. An extensible protocol architecture for application-specific networking. In Proceedings of the 1996 Winter USENIX Conference, pp. 55–64 (1996).

114 Floyd, S., Handley, M., Padhye, J., Widmer, J. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348 (Proposed Standard), 2008. URL `http://www.ietf.org/rfc/rfc5348.txt`.

115 Floyd, S., Henderson, T., Gurtov, A. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), 2004. URL `http://www.ietf.org/rfc/rfc3782.txt`. Obsoleted by RFC 6582.

116 Floyd, S., Kohler, E. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control. RFC 4341 (Proposed Standard), 2006. URL `http://www.ietf.org/rfc/rfc4341.txt`.

117 Floyd, S., Kohler, E., Padhye, J. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), 2006. URL `http://www.ietf.org/rfc/rfc4342.txt`. Updated by RFCs 5348, 6323.

118 Floyd, S., Paxson, V. Difficulties in simulating the Internet. IEEE/ACM Transactions on Networking (TON), 9 (4):pp. 392–403, 2001.

119 Fogarty, T.C. Varying the probability of mutation in the genetic algorithm. In Proceedings of the third international conference on Genetic algorithms, pp. 104–109, (Morgan Kaufmann Publishers Inc.1989).

120 Fonseca, R., Porter, G.M., Katz, R.H., Shenker, S., Stoica, I. IP options are not an option. Tech. rep., University of California at Berkeley, 2005.

121 Fortuna, C., Mohorcic, M. Trends in the development of communication networks: Cognitive networks. Computer Networks, 53 (9):pp. 1354 – 1376, 2009.

122 Gaïti, D., Pujolle, G., Salaun, M., Zimmermann, H. Autonomous Network Equipments. In I. Stavrakakis, M. Smirnov (eds.), Autonomic Communication, vol. 3854 of Lecture Notes in Computer Science, pp. 177–185, (Springer Berlin Heidelberg2006).

123 Gandomi, A.H., Yang, X.S., Alavi, A.H. Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. Engineering with Computers, 29 (1):pp. 17–35, 2013.

124 Ganek, A., Corbi, T.A. The dawning of the autonomic computing era. IBM Systems Journal, 42 (1):pp. 5–18, 2003.

125 Ganguly, N. Design and Analysis of a Bio-inspired Search Algorithm for Peer to Peer Networks. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. Moorsel, M. Steen (eds.), Self-star Properties in Complex Information Systems, vol. 3460 of Lecture Notes in Computer Science, pp. 358–372, (Springer Berlin Heidelberg2005).

126 García-Sánchez, P., González, J., Castillo, P., Arenas, M., Merelo-Guervós, J. Service oriented evolutionary algorithms. Soft Computing, pp. 1–17, 2013.

127 Gavalas, D., Greenwood, D., Ghanbari, M., O'Mahony, M. Advanced network monitoring applications based on mobile/intelligent agent technology. Computer Communications, 23 (8):pp. 720–730, 2000.

128 Gazis, V., Patouni, E., Alonistioti, N., Merakos, L. A survey of dynamically adaptable protocol stacks. Communications Surveys Tutorials, IEEE, 12 (1):pp. 3–23, 2010.

129 Gelenbe, E. Learning in the recurrent random neural network. Neural Computation, 5 (1):pp. 154–164, 1993.

130 —. Cognitive Routing in Packet Networks. In N. Pal, N. Kasabov, R. Mudi, P. Srimanta, S. Parui (eds.), Neural Information Processing, vol. 3316 of Lecture Notes in Computer Science, pp. 625–632, (Springer Berlin Heidelberg2004). URL `http://dx.doi.org/10.1007/978-3-540-30499-9_96`.

131 Gelenbe, E., Gellman, M., Lent, R., Liu, P., Su, P. Autonomous smart routing for network QoS. In Autonomic Computing, 2004. Proceedings. International Conference on, pp. 232–239 (2004).

132 Gelenbe, E., Lent, R., Montuori, A., Xu, Z. Cognitive packet networks: QoS and performance. In Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on, pp. 3–9 (2002).

133 Gelenbe, E., Lent, R., Nunez, A. Self-aware networks and QoS. Proceedings of the IEEE, 92 (9):pp. 1478–1489, 2004.

134 Gember, A., Grandl, R., Anand, A., Benson, T., Akella, A. Stratos: Virtual middleboxes as first-class entities. Tech. rep., Technical Report TR1771, University of Wisconsin-Madison, 2012.

135 GENI. `http://www.geni.net`.

136 Gharavi, H., Ban, K. Cross-layer feedback control for video communications via mobile ad-hoc networks. In Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th, vol. 5, pp. 2941–2945 Vol.5 (2003).

137 Goldberg, D.E. Genetic Algorithms in Search, Optimization, and Machine Learning, (Addison-Wesley1989).

138 Gonzalez, F., Dasgupta, D., Kozma, R. Combining negative selection and classification techniques for anomaly detection. In Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on, vol. 1, pp. 705–710 (2002).

139 Granelli, F., Pawelczak, P., Prasad, R., Subbalakshmi, K.P., Chandramouli, R., Hoffmeyer, J., Berger, H. Standardization and research in cognitive and dynamic spectrum access networks: IEEE SCC41 efforts and other activities. Communications Magazine, IEEE, 48 (1):pp. 71–79, 2010.

140 Grefenstette, J.J. Optimization of control parameters for genetic algorithms. Systems, Man and Cybernetics, IEEE Transactions on, 16 (1):pp. 122–128, 1986.

141 Gu, X., Klie, T., Wolf, L. A Proactive Policy-Based Management Approach Towards Autonomic Communications. In Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE, pp. 587–592 (2007).

142 Gu, Y., Grossman, R. Supporting Configurable Congestion Control in Data Transport Services. In Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, pp. 31–31 (2005).

143 Gutmann, P. Random Number Generation. http://www.cypherpunks.to/~peter/06_random.pdf, 2001.

144 —. Lessons learned in implementing and deploying crypto software. In Proc. USENIX Security Symp, pp. 315–325 (2002).

145 Guy, R.G., Heidemann, J.S., Mak, W., Page Jr, T.W., Popek, G.J., Rothmeier, D., et al. Implementation of the Ficus replicated file system. In USENIX Conference Proceedings, vol. 74, pp. 63–71, (Citeseer1990).

146 Hadjiantonis, A., Malatras, A., Pavlou, G. A context-aware, policy-based framework for the management of MANETs. In Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop on, pp. 10 pp.–34 (2006).

147 HAGGLE. http://www.haggleproject.org.

148 Haigh, K., Varadarajan, S., Tang, C.Y. Automatic Learning-based MANET Cross-Layer Parameter Configuration. In Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on, pp. 84–84 (2006).

149 Han, D., Anand, A., Dogar, F., Li, B., Lim, H., Machado, M., Mukundan, A., Wu, W., Akella, A., Andersen, D.G., et al. XIA: Efficient support for evolvable internetworking. Proc. 9th USENIX NSDI, 2012.

150 Han, J., Kamber, M., Pei, J. Data Mining: Concepts and Techniques, (Morgan Kaufmann2006).

151 Han, Q., Gutierrez-Nolasco, S., Venkatasubramanian, N. Reflective middleware for integrating network monitoring with adaptive object messaging. Network, IEEE, 18 (1):pp. 56–65, 2004.

152 Handley, M., Kohler, E., Ghosh, A., Hodson, O., Radoslavov, P. Designing extensible IP router software. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05, pp. 189–202, (USENIX Association, Berkeley, CA, USA2005).

153 Harai, H., Fujikawa, K., Kafle, V.P., Miyazawa, T., Murata, M., Ohnishi, M., Ohta, M., Umezawa, T. Design Guidelines for New Generation Network Architecture. IEICE TRANSACTIONS on Communications, E93-B (3):pp. 462–465, 2010.

154 Harmer, P., Williams, P., Gunsch, G., Lamont, G. An artificial immune system

architecture for computer security applications. Evolutionary Computation, IEEE Transactions on, 6 (3):pp. 252–280, 2002.

155 Harrington, D., Presuhn, R., Wijnen, B. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411 (Standard), 2002. URL http://www.ietf.org/rfc/rfc3411.txt. Updated by RFCs 5343, 5590.

156 Hayden, M.G. The Ensemble System. Ph.D. thesis, Cornell University, 1998.

157 Haykin, S. Cognitive radio: brain-empowered wireless communications. Selected Areas in Communications, IEEE Journal on, 23 (2):pp. 201–220, 2005.

158 He, E., Vicat-Blanc, P., Welzl, M. A Survey of Transport Protocols other than Standard TCP. In Global Grid Forum (2005).

159 Hicks, M., Kakkar, P., Moore, J.T., Gunter, C.A., Nettles, S. PLAN: A Packet Language for Active Networks. SIGPLAN Not., 34 (1):pp. 86–93, 1998.

160 Hopcroft, J.E., Motwani, R., Ullman, J.D. Introduction to automata theory, languages, and computation, vol. 2, (Addison-wesley Reading, MA1979).

161 Horrocks, I. DAML+ OIL: a reason-able web ontology language. Advances in Database Technology—EDBT 2002, pp. 103–116, 2002.

162 Horvitz, D.G., Thompson, D.J. A generalization of sampling without replacement from a finite universe. Journal of the American Statistical Association, 47 (260):pp. 663–685, 1952.

163 Hunt, J.E., Cooke, D.E. Learning using an artificial immune system. J. Netw. Comput. Appl., 19 (2):pp. 189–212, 1996. URL http://dx.doi.org/10.1006/jnca.1996.0014.

164 Hutchinson, N.C., Peterson, L.L. The X-Kernel: An Architecture for Implementing Network Protocols. IEEE Trans. Softw. Eng., 17 (1):pp. 64–76, 1991.

165 IEEE. Draft STANDARD for Information Technology-Telecommunications and information exchange between systems-Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment : Enhancements for Higher Throughput. IEEE Draft Std P802.11n/D2.00, February 2007, 2007.

166 —. IEEE Standard for Architectural Building Blocks Enabling Network-Device Distributed Decision Making for Optimized Radio Resource Usage in Heterogeneous Wireless Access Networks. IEEE Std 1900.4-2009, pp. C1–119, 2009.

167 —. IEEE Standard for Policy Language Requirements and System Architectures for Dynamic Spectrum Access Systems. IEEE Std 1900.5-2011, pp. 1–51, 2012.

168 Imai, P., Lamparter, B., Liebsch, M. Policy-Based Device and Mobility Management. In MOBILWARE, pp. 101–114 (2009).

169 Imai, P., Lopez, Y., Legendre, F., May, M. The SAC Gateway: Federating the Future Internet of Wireless Clouds. In ACM MobiCom-MobiHoc'10 Poster Session (2010).

170 Imai, P., Tschudin, C. Practical Online Network Stack Evolution. In Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on, pp. 34–41 (2010).

171 Jacobson, V., Braden, R., Borman, D. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), 1992. URL http://www.ietf.org/rfc/rfc1323.txt.

172 Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.L. Networking named content. In Proceedings of the 5th international conference on Emerging networking experiments and technologies, CoNEXT '09, pp. 1–12, (ACM, New York, NY, USA2009).

173 Jaganathan, R., Underwood, K., Sass, R. A Configurable Network Protocol for Cluster Based Communications using Modular Hardware Primitives on an Intelligent NIC. In Supercomputing, 2003 ACM/IEEE Conference, pp. 22–22 (2003).

174 Jain, M., Dovrolis, C. Pathload: A Measurement Tool for End-to-End Available Bandwidth. In In Proceedings of Passive and Active Measurements (PAM) Workshop, pp. 14–25 (2002).

175 Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., Towsley, D. Inferring TCP connection characteristics through passive measurements. In INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, vol. 3, pp. 1582–1592 vol.3 (2004).

176 Jeon, S.W., Devroye, N., Vu, M., Chung, S.Y., Tarokh, V. Cognitive Networks Achieve Throughput Scaling of a Homogeneous Network. Information Theory, IEEE Transactions on, 57 (8):pp. 5103–5115, 2011.

177 de Jong, K.A. An analysis of the behavior of a class of genetic adaptive systems. Ph.D. thesis, University of Michigan, 1975.

178 Kaelbling, L.P., Littman, M.L., Moore, A.W. Reinforcement learning: A survey. Journal of Artificial Intelligence Research, 4:pp. 237–285, 1996.

179 Kaiser, G., Gross, P., Kc, G., Parekh, J., Valetto, G. An approach to autonomizing legacy systems. In in Workshop on Self-Healing, Adaptive and Self-MANaged Systems (2002).

180 Kappler, C., Mendes, P., Prehofer, C., Pöyhönen, P., Zhou, D. A Framework for Self-organized Network Composition. In M. Smirnov (ed.), Autonomic Communication, vol. 3457 of Lecture Notes in Computer Science, pp. 139–151, (Springer Berlin Heidelberg2005).

181 Karaboga, D., Akay, B. A comparative study of artificial bee colony algorithm. Applied Mathematics and Computation, 214 (1):pp. 108–132, 2009.

182 Karaboga, D., Basturk, B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of global optimization, 39 (3):pp. 459–471, 2007.

183 Karsten, M., Keshav, S., Prasad, S., Beg, M. An Axiomatic Basis for Communication. ACM SIGCOMM Computer Communication Review, 37 (4):pp. 217–228, 2007.

184 Kashiwagi, A., Urabe, I., Kaneko, K., Yomo, T. Adaptive Response of a Gene Network to Environmental Changes by Fitness-Induces Attractor Selection. PLoS One, 1 (1):p. e49, 2006.

185 Kass, R.E., Wasserman, L. A Reference Bayesian Test for Nested Hypotheses and its Relationship to the Schwarz Criterion. Journal of the American Statistical Association, 90 (431):pp. 928–934, 1995.

186 Keeney, R.L., Raiffa, H. Decisions with multiple objectives: preferences and value trade-offs, (Cambridge University Press1993).

187 Keller, A., Hossmann, T., May, M., Bouabene, G., Jelger, C., Tschudin, C. A system architecture for evolving protocol stacks. In Computer Communications and Networks, 2008. ICCCN'08. Proceedings of 17th International Conference on, pp. 1–7, (IEEE2008).

188 Kelly, T., et al. Utility-directed allocation. In First Workshop on Algorithms and Architectures for Self-Managing Systems, vol. 20 (3) (2003).

189 Kephart, J., Das, R. Achieving Self-Management via Utility Functions. Internet Computing, IEEE, 11 (1):pp. 40–48, 2007.

190 Kephart, J.O. A Biologically Inspired Immune System for Computers. In In Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp. 130–139, (MIT Press1994).

191 Kephart, J.O., Chess, D.M. The Vision of Autonomic Computing. Computer, 36 (1):pp. 41–50, 2003.

192 Kilcher, Y. Autonomous Transport Protocol Optimization using DCCP, 2012.

193 Kim, K.W., Lorenz, P., Lee, M. A new tuning maximum congestion window for improving TCP performance in MANET. In Systems Communications, 2005. Proceedings, pp. 73–78 (2005).

194 Kirkpatrick, S., Jr., D.G., Vecchi, M.P. Optimization by simulated annealing. Science, 220 (4598):pp. 671–680, 1983.

195 Kohler, E., Handley, M., Floyd, S. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), 2006. URL http://www.ietf.org/rfc/rfc4340.txt. Updated by RFCs 5595, 5596, 6335.

196 Kohler, E., Kaashoek, M.F., Montgomery, D.R. A readable TCP in the Prolac protocol language. SIGCOMM Comput. Commun. Rev., 29 (4):pp. 3–13, 1999.

197 Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F. The Click Modular Router. ACM Transactions on Computer Systems, 18 (3):pp. 263–297, 2000.

198 Kompella, R.R., Levchenko, K., Snoeren, A.C., Varghese, G. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. SIGCOMM Comput. Commun. Rev., 39 (4):pp. 255–266, 2009.

199 Koponen, T., Chawla, M., Chun, B.G., Ermolinskiy, A., Kim, K.H., Shenker, S., Stoica, I. A Data-Oriented (and Beyond) Network Architecture. SIGCOMM Comput. Commun. Rev., 37 (4):pp. 181–192, 2007.

200 Kozma, R., Kitamura, M., Sakuma, M., Yokoyama, Y. Anomaly detection by neural network models and statistical time series analysis. In Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on, vol. 5, pp. 3207–3210 vol.5 (1994).

201 Kramer, O. Evolutionary self-adaptation: a survey of operators and strategy parameters. Evolutionary Intelligence, 3:pp. 51–65, 2010.

202 Krasnyansky, M. Universal TUN/TAP Driver – Virtual Point-to-Point(TUN) and Ethernet(TAP) devices. URL http://vtun.sourceforge.net/tun/.

203 Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y. Sketch-based change detection: methods, evaluation, and applications. In Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, pp. 234–247, (ACM2003).

204 Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B. OceanStore: an architecture for global-scale persistent storage. SIGPLAN Not., 35 (11):pp. 190–201, 2000.

205 Kuo, A. An optimal control model for analyzing human postural balance. Biomedical Engineering, IEEE Transactions on, 42 (1):pp. 87–101, 1995.

206 Lakhina, A., Crovella, M., Diot, C. Mining anomalies using traffic feature distributions. SIGCOMM Comput. Commun. Rev., 35 (4):pp. 217–228, 2005.

207 Laredo, J., Castillo, P., Mora, A., Merelo, J. Evolvable agents, a fine grained approach for distributed evolutionary

computing: walking towards the peer-to-peer computing frontiers. Soft Computing, 12:pp. 1145–1156, 2008.

208 Lassila, O., Swick, R.R., et al. Resource description framework (RDF) model and syntax specification, 1998.

209 Lavalle, S.M. Rapidly-Exploring Random Trees: A New Tool for Path Planning. self-published, 1998.

210 LE THANH, M., HASEGAWA, G., MURATA, M. ImTCP: TCP with an inline network measurement mechanism. IEIC Technical Report (Institute of Electronics, Information and Communication Engineers), 104 (73):pp. 37–42, 2004.

211 Lee, G., Bauer, S., Faratin, P., Wroclawski, J. Learning User Preferences for Wireless Services Provisioning. In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1, AAMAS '04, pp. 480–487, (IEEE Computer Society, Washington, DC, USA2004).

212 Lee, G., Faratin, P., Bauer, S., Wroclawski, J. A user-guided cognitive agent for network service selection in pervasive computing environments. In Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on, pp. 219–228 (2004).

213 Lee, M., Goldberg, S., Kompella, R.R., Varghese, G. Fine-grained latency and loss measurements in the presence of reordering. In Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, SIGMETRICS '11, pp. 329–340, (ACM, New York, NY, USA2011).

214 Leibnitz, K. Symbiotic Multi-Path Routing with Attractor Selection. Communications of the ACM, 49 (3):pp. 62–67, 2006.

215 Levis, P., Culler, D. Maté: A tiny virtual machine for sensor networks. In ACM Sigplan Notices, vol. 37 (10), pp. 85–95, (ACM2002).

216 Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. TinyOS: An operating system for sensor networks. Ambient intelligence, 35, 2005.

217 Lewis, D., Keeney, J., O'Sullivan, D., Guo, S. Towards a Managed Extensible Control Plane for Knowledge-Based Networking. In R. State, S. Meer, D. O'Sullivan, T. Pfeifer (eds.), Large Scale Management of Distributed Systems, vol. 4269 of Lecture Notes in Computer Science, pp. 98–111, (Springer Berlin Heidelberg2006).

218 Lindley, D.V., Lindley, D. Bayesian statistics: A review, vol. 2, (SIAM1972).

219 Lindörfer, F. Online Protocol Selection through Machine Learning Algorithms, 2010.

220 Liotta, A., Pavlou, G., Knight, G. Exploiting agent mobility for large-scale network monitoring. Network, IEEE, 16 (3):pp. 7–15, 2002.

221 uan M. Estévez-Tapiador, annd Jesús E. Dı☒z-Verdejo, P.G.T. Measuring normality in HTTP traffic for anomaly-based intrusion detection. Computer Networks, 45 (2):pp. 175–193, 2004.

222 Macedo, D., Dos Santos, A., Nogueira, J., Pujolle, G. A distributed information repository for autonomic context-aware MANETs. Network and Service Management, IEEE Transactions on, 6 (1):pp. 45–55, 2009.

223 Macedo, D., Dos Santos, A., Pujolle, G., Nogueira, J. MANKOP: A Knowledge Plane for wireless ad hoc networks. In Network Operations and Management Symposium, 2008. NOMS 2008. IEEE, pp. 706–709 (2008).

224 Macedo, D.F., Santos, A.L., Nogueira, J.M.S., Pujolle, G. A Knowledge Plane for Autonomic Context-Aware Wireless Mobile Ad Hoc Networks. In G. Pavlou,

T. Ahmed, T. Dagiuklas (eds.), Management of Converged Multimedia Networks and Services, vol. 5274 of Lecture Notes in Computer Science, pp. 1–13, (Springer Berlin Heidelberg2008).

225 Maeda, C., Bershad, B.N. Protocol service decomposition for high-performance networking. SIGOPS Oper. Syst. Rev., 27 (5):pp. 244–255, 1993.

226 Magedanz, T., Rothermel, K., Krause, S. Intelligent agents: an emerging technology for next generation telecommunications? In INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE, vol. 2, pp. 464–472 vol.2 (1996).

227 Mahajan, R., Spring, N., Wetherall, D., Anderson, T. User-level internet path diagnosis. SIGOPS Oper. Syst. Rev., 37 (5):pp. 106–119, 2003.

228 Mamei, M., Zambonelli, F. Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol., 18 (4):pp. 15:1–15:56, 2009.

229 Mamei, M., Zambonelli, F., Leonardi, L. Cofields: a physically inspired approach to motion coordination. Pervasive Computing, IEEE, 3 (2):pp. 52–61, 2004.

230 Mandelbrot, B.B. The fractal geometry of nature, (Times Books1983).

231 Martin, D., Völker, L., Zitterbart, M. A flexible framework for Future Internet design, assessment, and operation. Computer Networks, 55 (4):pp. 910 – 918, 2011.

232 Massie, M.L., Chun, B.N., Culler, D.E. The Ganglia Distributed Monitoring System: Design, implementation, and experience. Parallel Computing, 30 (7):pp. 817–840, 2004.

233 Mathis, M., Mahdavi, J., Floyd, S., Romanow, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), 1996. URL http://www.ietf.org/rfc/rfc2018.txt.

234 McCanne, S., Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, pp. 2–2, (USENIX Association1993).

235 McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J. OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38 (2):pp. 69–74, 2008.

236 McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H. Composing adaptive software. Computer, 37 (7):pp. 56–64, 2004.

237 McQuarrie, A.D., Tsai, C.L. Regression and time series model selection, vol. 43, (World Scientific Singapore1998).

238 Mena, S., Cuvellier, X., Gregoire, C., Schiper, A. Appia vs. Cactus: comparing protocol composition frameworks. In Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on, pp. 189–198 (2003).

239 Meyer, T., Yamamoto, L., Tschudin, C. An Artificial Chemistry for Networking. BioInspired Computing and Communication, 5151 (Biowire 2007):pp. 45–57, 2008. URL http://www.springerlink.com/index/d03671k161t46v70.pdf.

240 Miao, K.X., Schulzrinne, H., Singh, V.K., Deng, Q. Distributed Self Fault-Diagnosis for SIP Multimedia Applications. In Real-Time Mobile Multimedia Services, pp. 187–190, (Springer2007).

241 Mills, D. Measured performance of the Network Time Protocol in the Internet system, 1989. URL http://www.dspace.cam.ac.uk/handle/1810/13951.

242 Mills, D., Martin, J., Burbank, J., Kasch, W. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), 2010. URL http://www.ietf.org/rfc/rfc5905.txt.

243 Miranda, H., Pinto, A., Rodrigues, L. Appia, a flexible protocol kernel supporting multiple coordinated channels. In Distributed Computing Systems, 2001. 21st International Conference on., pp. 707–710 (2001).

244 Mitola, J. Cognitive Radio Architecture Evolution. Proceedings of the IEEE, 97 (4):pp. 626–641, 2009.

245 Mitola, J., Maguire, G.Q., J. Cognitive radio: making software radios more personal. Personal Communications, IEEE, 6 (4):pp. 13–18, 1999.

246 MobilityFirst. http://mobilityfirst.winlab.rutgers.edu.

247 Mockapetris, P. Domain names - implementation and specification. RFC 1035 (Standard), 1987. URL http://www.ietf.org/rfc/rfc1035.txt. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

248 Mohri, M., Rostamizadeh, A., Talwalkar, A. Foundations of Machine Learning, (The MIT Press2012).

249 Montana, D., Hussain, T. Adaptive reconfiguration of data networks using genetic algorithms. Applied Soft Computing, 4 (4):pp. 433 – 444, 2004.

250 Mori, N. Adaptation to a Dynamical Environment by Means of the Environment Identifying Genetic Algorithm. In Industrial Electronics Society, 2000. IECON 2000. 26th Annual Confjerence of the IEEE, vol. 4, pp. 2953–2958 vol.4 (2000).

251 Mowbray, T.J., Zahavi, R. The essential CORBA: systems integration using distributed objects, (John Wiley & Sons, Inc.1995).

252 Mühleisen, H., Augustin, A., Walther, T., Harasic, M., Teymourian, K., Tolksdorf, R. A self-organized semantic storage service. In Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services, iiWAS '10, pp. 357–364, (ACM, New York, NY, USA2010).

253 Muhugusa, M., Di Marzo, G., Tschudin, C., Solana, E., Harms, J. Implementation and interpretation of protocols in the COMSCRIPT environment. In Communications, 1995. ICC '95 Seattle, 'Gateway to Globalization', 1995 IEEE International Conference on, vol. 1, pp. 379–384 vol.1 (1995).

254 Murata, N., Yoshizawa, S., Amari, S.I. Network information criterion-determining the number of hidden units for an artificial neural network model. Neural Networks, IEEE Transactions on, 5 (6):pp. 865–872, 1994.

255 Muthukrishnan, S. Data streams: Algorithms and applications, (Now Publishers Inc2005).

256 Nahm, K., Helmy, A., Jay Kuo, C.C. TCP over multihop 802.11 networks: Issues and performance enhancement. In Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing, pp. 277–287, (ACM2005).

257 Named Data Networking Project. http://www.named-data.net.

258 NEBULA. http://nebula.cis.upenn.edu.

259 Neglia, G., Reina, G. Evaluating Activator-Inhibitor Mechanisms for Sensor Coordination. In Bio-Inspired Models of Network, Information and Computing Systems, 2007. Bionetics 2007. 2nd, pp. 129–133 (2007).

260 von Neumann, J., Morgenstern, O. The theory of games and economic behavior, (Princeton university press1947).

261 Nichols, K., Blake, S., Baker, F., Black, D. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), 1998. URL http://www.ietf.org/rfc/rfc2474.txt. Updated by RFCs 3168, 3260.

262 Nolan, K., Sutton, P., Doyle, L. An Encapsulation for Reasoning, Learning, Knowledge Representation, and Reconfiguration Cognitive Radio Elements. In Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on, pp. 1–5 (2006).

263 Nolle, L., Goodyear, A., Hopgood, A.A., Picton, P.D., Braithwaite, N.S. On step width adaptation in simulated annealing for continuous parameter optimisation. In Computational Intelligence. Theory and Applications, pp. 589–598, (Springer2001).

264 Nowak, M.A. Evolutionary Dynamics: Exploring the Equations of Life, (Belknap Press2006).

265 Ochoa, G. Error thresholds in genetic algorithms. Evolutionary computation, 14 (2):pp. 157–182, 2006.

266 Ogel, F., Patarin, S., Piumarta, I., Folliot, B. C/SPAN: a self-adapting web proxy cache. In Autonomic Computing Workshop. 2003. Proceedings of the, pp. 178–185, (IEEE2003).

267 O'Malley, S.W., Peterson, L.L. A dynamic network architecture. ACM Trans. Comput. Syst., 10 (2):pp. 110–143, 1992.

268 Pahdye, J., Floyd, S. On inferring TCP behavior. SIGCOMM Comput. Commun. Rev., 31 (4):pp. 287–298, 2001.

269 Parashar, M., Hariri, S. Autonomic computing: An overview. Unconventional Programming Paradigms, pp. 97–97, 2005.

270 Parlos, A., Menon, S., Atiya, A. An algorithmic approach to adaptive state filtering using recurrent neural networks. Neural Networks, IEEE Transactions on, 12 (6):pp. 1411–1432, 2001.

271 Partridge, C., Kastenholz, F. Technical Criteria for Choosing IP The Next Generation (IPng). RFC 1726 (Informational), 1994. URL http://www.ietf.org/rfc/rfc1726.txt.

272 Patarin, S., Patarin, S., Makpangou, M., Makpangou, M., Pat, S. Pandora: A Flexible Network Monitoring Platform. In In Proceedings of the USENIX 2000 Annual Technical Conference, pp. 200–0 (2000).

273 Patel, P., Whitaker, A., Wetherall, D., Lepreau, J., Stack, T. Upgrading transport protocols using untrusted mobile code. In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 1–14, (ACM, New York, NY, USA2003).

274 Paxson, V. End-to-end routing behavior in the Internet. SIGCOMM Comput. Commun. Rev., 26 (4):pp. 25–38, 1996.

275 Paxson, V., Allman, M. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), 2000. URL http://www.ietf.org/rfc/rfc2988.txt. Obsoleted by RFC 6298.

276 Paxson, V., Floyd, S. Why we don't know how to simulate the Internet. In Proceedings of the 29th conference on Winter simulation, pp. 1037–1044, (IEEE Computer Society1997).

277 Pelleg, D., Moore, A.W. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00, pp. 727–734, (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA2000). URL http://dl.acm.org/citation.cfm?id=645529.657808.

278 Perkins, C.E. Mobile IP. Communications Magazine, IEEE, 35 (5):pp. 84–99, 1997.

279 Petrova, M., Mahonen, P., Riihijarvi, J. Evolution of Radio Resource Management: A Case for Cognitive Resource Manager with VPI. In Communications, 2007. ICC '07. IEEE International Conference on, pp. 6471–6475 (2007).

280 Pitchaimani, M., Ewy, B., Evans, J. Evaluating Techniques for Network Layer Independence in Cognitive Networks. In Communications, 2007. ICC '07. IEEE International Conference on, pp. 6527–6531 (2007).

281 Plagemann, T., Plattner, B., Vogt, M., Walter, T. A model for dynamic configuration of light-weight protocols. In Distributed Computing Systems, 1992., Proceedings of the Third Workshop on Future Trends of, pp. 100–106 (1992).

282 Plagemann, T., Vogt, M., Plattner, B., Walter, T. Modules as building blocks for protocol configuration. In Network Protocols, 1993. Proceedings., 1993 International Conference on, pp. 106–113 (1993).

283 PlanetLab. http://www.planet-lab.org.

284 Plummer, D. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Standard), 1982. URL http://www.ietf.org/rfc/rfc826.txt. Updated by RFCs 5227, 5494.

285 Popa, L., Ghodsi, A., Stoica, I. HTTP as the Narrow Waist of the Future Internet. In Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, p. 6, (ACM2010).

286 Postel, J. User Datagram Protocol. RFC 768 (Standard), 1980. URL http://www.ietf.org/rfc/rfc768.txt.

287 —. Internet Control Message Protocol. RFC 792 (Standard), 1981. URL http://www.ietf.org/rfc/rfc792.txt. Updated by RFCs 950, 4884, 6633.

288 —. Internet Protocol. RFC 791 (Standard), 1981. URL http://www.ietf.org/rfc/rfc791.txt. Updated by RFCs 1349, 2474.

289 —. Transmission Control Protocol. RFC 793 (Standard), 1981. URL http://www.ietf.org/rfc/rfc793.txt. Updated by RFCs 1122, 3168, 6093, 6528.

290 Postel, J., Reynolds, J. File Transfer Protocol. RFC 959 (Standard), 1985. URL http://www.ietf.org/rfc/rfc959.txt. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

291 Pujolle, G., Chaouchi, H. An autonomic-oriented architecture for wireless sensor networks. Annales Des Télécommunications, 60:pp. 819–830, 2005.

292 Pujolle, G., Chaouchi, H., Gaïti, D. Beyond TCP/IP: A context-aware architecture. Network Control and Engineering for QoS, Security and Mobility, III, pp. 337–346, 2005.

293 Ramabhadran, S., Varghese, G. Efficient implementation of a statistics counter architecture. SIGMETRICS Perform. Eval. Rev., 31 (1):pp. 261–271, 2003.

294 Ramakrishnan, K., Floyd, S., Black, D. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), 2001. URL http://www.ietf.org/rfc/rfc3168.txt. Updated by RFCs 4301, 6040.

295 Ramming, C. Cognitive Networks. DARPA Tech, 2004.

296 Ramos-Munoz, J.J., Yamamoto, L., Tschudin, C. Serial Experiments Online. In ACM SIGCOMM Computer Communication Review, vol. 38 (2), pp. 31–42, (ACM2008).

297 Rasmussen, J. Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models. Systems, Man and Cybernetics, IEEE Transactions on, SMC-13 (3):pp. 257–266, 1983.

298 Raychaudhuri, D., Mandayam, N.B., Evans, J.B., Ewy, B.J., Seshan, S., Steenkiste, P. CogNet: an architectural foundation for experimental cognitive radio networks within the future internet. In Proceedings of first ACM/IEEE international workshop on Mobility in the evolving internet architecture, MobiArch '06, pp. 11–16, (ACM, New York, NY, USA2006).

299 Razzaque, M., Dobson, S., Nixon, P. Enhancement of Self-organisation in Wireless Networking through a Cross-Layer Approach. In J. Zheng, S. Mao, S. Midkiff, H. Zhu (eds.), Ad Hoc Networks, vol. 28 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 144–159, (Springer Berlin Heidelberg2010).

300 Razzaque, M., Nixon, P., Dobson, S. Demonstrating the feasibility of an Autonomic Communication- Targeted Cross-Layer Architecture. In Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on, pp. 67–72 (2006).

301 Razzaque, M.A., Dobson, S.A., Nixon, P. A Cross-Layer Architecture for Autonomic Communications. In Autonomic Networking, pp. 25–35 (2006).

302 van Renesse, R., Birman, K.P., Maffeis, S. Horus: a flexible group communication system. Commun. ACM, 39 (4):pp. 76–83, 1996.

303 Ritchie, D.M. A Stream Input-Output System. AT&T Bell Laboratories Technical Journal, 63:pp. 311–324, 1984.

304 Rizzo, L. Dummynet: A Simple Approach to the Evaluation of Network Protocols. ACM Computer Communication Review, 27:pp. 31–41, 1997.

305 Robles, J., Tromer, S., Hidalgo, J., Lehnert, R. A high configurable protocol for indoor localization systems. In Indoor Positioning and Indoor Navigation (IPIN), 2011 International Conference on, pp. 1–7 (2011).

306 Rudolph, G. Evolutionary search for minimal elements in partially ordered finite sets. In Evolutionary Programming VII, pp. 345–353, (Springer1998).

307 Russell, L.W., Morgan, S.P., Chron, E. Clockwork: A new movement in autonomic systems. IBM Systems Journal, 42 (1):pp. 77–84, 2003.

308 Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M., Edwards, D.D. Artificial Intelligence: A Modern Approach, vol. 74, (Prentice hall Englewood Cliffs1995).

309 Saffre, F., Blok, H. "SelfService": a theoretical protocol for autonomic distribution of services in P2P communities. In Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the, pp. 528–534 (2005).

310 Sarafijanovic, S., Le Boudec, J.Y. An artificial immune system approach with secondary response for misbehavior detection in mobile ad hoc networks. Neural Networks, IEEE Transactions on, 16 (5):pp. 1076–1087, 2005.

311 Savage, S. Sting: a TCP-based network measurement tool. In Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems, pp. 71–79 (1999).

312 Schwartz, B., Jackson, A., Strayer, W., Zhou, W., Rockwell, R., Partridge, C. Smart Packets for active networks. In Open Architectures and Network Programming Proceedings, 1999. OPENARCH '99. 1999 IEEE Second Conference on, pp. 90–97 (1999).

313 Sesum-Cavic, V., Kuhn, E. A Swarm Intelligence Appliance to the Construction of an Intelligent Peer-to-Peer Overlay Network. In Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on, pp. 1028–1035 (2010).

314 Sherwood, R., Spring, N. A platform for unobtrusive measurements on PlanetLab. In Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems - Volume 3, WORLDS'06, pp. 2–2, (USENIX Association, Berkeley, CA, USA2006).

315 Shieh, A., Myers, A.C., Sirer, E.G. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pp. 175–188, (USENIX Association2005).

316 Sifalakis, M., Fry, M., Hutchison, D. Event detection and correlation for network environments. Selected Areas in Communications, IEEE Journal on, 28 (1):pp. 60–69, 2010.

317 Sifalakis, M., Louca, A., Mauthe, A., Peluso, L., Zseby, T. A Functional Composition Framework for Autonomic Network Architectures. In Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE, pp. 328–334 (2008).

318 Simon, H.A., Simon, P.A. Trial and Error Search in Solving Difficult Problems: Evidence from the Game of Chess. Behavioral Science, 7 (4):pp. 425–429, 1962.

319 Smirnov, M. Autonomic Communication—Research Agenda for a new Communication Paradigm. Company whitepaper. Fraunhofer Institute for Open Communication Systems, Berlin, Germany, 2004.

320 Smit, S., Eiben, A. Parameter Tuning of Evolutionary Algorithms: Generalist vs. Specialist. In C. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, C.K. Goh, J. Merelo, F. Neri, M. Preuß, J. Togelius, G. Yannakakis (eds.), Applications of Evolutionary Computation, vol. 6024 of Lecture Notes in Computer Science, pp. 542–551, (Springer Berlin Heidelberg2010).

321 —. Using Entropy for Parameter Analysis of Evolutionary Algorithms. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss (eds.), Experimental Methods for the Analysis of Optimization Algorithms, pp. 287–310, (Springer Berlin Heidelberg2010).

322 Smit, S.K., Eiben, A.E. Comparing parameter tuning methods for evolutionary algorithms. In Evolutionary Computation, 2009. CEC'09. IEEE Congress on, pp. 399–406, (IEEE2009).

323 Smith, J., Fogarty, T.C. Self adaptation of mutation rates in a steady state genetic algorithm. In Evolutionary Computation, 1996., Proceedings of IEEE International Conference on, pp. 318–323, (IEEE1996).

324 Soekris Engineering, Inc. URL `http://soekris.com/products/net6501.html`.

325 Sommers, J., Barford, P. An active measurement system for shared environments. In Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, IMC '07, pp. 303–314, (ACM, New York, NY, USA2007).

326 Song, H.H., Qiu, L., Zhang, Y. NetQuest: a flexible framework for large-scale network measurement. SIGMETRICS Perform. Eval. Rev., 34 (1):pp. 121–132, 2006.

327 Spring, N., Wetherall, D., Anderson, T. Reverse engineering the Internet. SIGCOMM Comput. Commun. Rev., 34 (1):pp. 3–8, 2004.

328 Sridhar, P., Nanayakkara, T., Madni, A., Jamshidi, M. Dynamic power management of an embedded sensor network based on actor-critic reinforcement based learning. In Information and Automation for Sustainability, 2007. ICIAFS 2007. Third International Conference on, pp. 76–81 (2007).

329 Srisuresh, P., Egevang, K. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), 2001. URL `http://www.ietf.org/rfc/rfc3022.txt`.

330 Srivastava, V., Motani, M. Cross-layer design: a survey and the road ahead. Communications Magazine, IEEE, 43 (12):pp. 112–119, 2005.

331 Sterritt, R., Bustard, D. Towards an autonomic computing environment. In Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on, pp. 694–698 (2003).

332 Stoica, I., Adkins, D., Zhuang, S., Shenker, S., Surana, S. Internet indirection infrastructure. SIGCOMM Comput. Commun. Rev., 32 (4):pp. 73–86, 2002.

333 Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev., 31 (4):pp. 149–160, 2001.

334 Stone, P., Veloso, M. Multiagent systems: A survey from a machine learning perspective. Autonomous Robots, 8 (3):pp. 345–383, 2000.

335 Stoy, K., Nagpal, R. Self-repair through scale independent self-reconfiguration. In Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, vol. 2, pp. 2062–2067 vol.2 (2004).

336 —. Self-reconfiguration using directed growth. Distributed Autonomous Robotic Systems 6, pp. 3–12, 2007.

337 Strassner, J., OFoghlu, M., Donnelly, W., Agoulmine, N. Beyond the Knowledge Plane: An Inference Plane to Support the Next Generation Internet. In Global Information Infrastructure Symposium, 2007. GIIS 2007. First International, pp. 112–119 (2007).

338 Strassner, J., Samudrala, S., Cox, G., Liu, Y., Jiang, M., Zhang, J., Meer, S.v.d., Foghlú, M.Ó., Donnelly, W. The Design of a New Context-Aware Policy Model for Autonomic Networking. In Autonomic Computing, 2008. ICAC '08. International Conference on, pp. 119–128 (2008).

339 Su, J., Scott, J., Hui, P., Crowcroft, J., De Lara, E., Diot, C., Goel, A., Lim, M.H., Upton, E. Haggle: Seamless networking for mobile applications. In UbiComp 2007: Ubiquitous Computing, pp. 391–408, (Springer2007).

340 Sutton, P., Doyle, L., Nolan, K. A Reconfigurable Platform for Cognitive Networks. In Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on, pp. 1–5 (2006).

341 Sutton, P., Lotze, J., Lahlou, H., Fahmy, S., Nolan, K., Ozgul, B., Rondeau, T., Noguera, J., Doyle, L. Iris: an architecture for cognitive radio networking testbeds. Communications Magazine, IEEE, 48 (9):pp. 114–122, 2010.

342 Sutton, P., Lotze, J., Lahlou, H., Ozgul, B., Fahmy, S., Nolan, K., Noguera, J., Doyle, L. Multi-platform demonstrations using the Iris architecture for cognitive radio network testbeds. In Cognitive Radio Oriented Wireless Networks & Communications (CROWNCOM), 2010 Proceedings of the Fifth International Conference on, pp. 1–5, (IEEE2010).

343 Sutton, R.S., Barto, A.G. Reinforcement learning: An introduction, vol. 1, (Cambridge Univ Press1998).

344 Szaniawski, K. The value of perfect information. Synthese, 17 (1):pp. 408–424, 1967.

345 Tadayoni, R., Henten, A. Transition from IPv4 to IPv6. In 23rd European Regional ITS Conference, Vienna 2012, (International Telecommunications Society (ITS)2012).

346 Tateson, R. Self-organising pattern formation: fruit flies and cell phones. In Parallel problem solving from nature—PPSN V, pp. 732–741, (Springer1998).

347 Tennenhouse, D.L., Smith, J.M., Sincoskie, W.D., Wetherall, D.J., Minden, G.J. A Survey of active network Research. IEEE Communications Magazine, 35 (1):pp. 80–86, 1997.

348 Tennenhouse, D.L., Wetherall, D.J. Towards an active network architecture. SIGCOMM Comput. Commun. Rev., 37 (5):pp. 81–94, 2007.

349 Thekkath, C.A., Nguyen, T.D., Moy, E., Lazowska, E.D. Implementing network protocols at user level. IEEE/ACM Transactions on Networking (TON), 1 (5):pp. 554–565, 1993.

350 Thomas, R., DaSilva, L., MacKenzie, A. Cognitive networks. In New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on, pp. 352–360 (2005).

351 Thomas, R., DaSilva, L., Marathe, M., Wood, K. Critical Design Decisions for Cognitive Networks. In Communications, 2007. ICC '07. IEEE International Conference on, pp. 3993–3998 (2007).

352 Tierney, B. TCP tuning guide for distributed applications on wide area networks. USENIX & SAGE Login, 26 (1):pp. 33–39, 2001.

353 Tolksdorf, R., Menezes, R. Using swarm intelligence in linda systems. Engineering Societies in the Agents World IV, pp. 519–519, 2004.

354 Touch, J., Baldine, I., Dutta, R., Finn, G.G., Ford, B., Jordan, S., Massey, D., Matta, A., Papadopoulos, C., Reiher, P., Rouskas, G. A Dynamic Recursive Unified Internet Design (DRUID). Computer Networks, 55 (4):pp. 919 – 935, 2011.

355 Touch, J.D., Wang, Y.S., Pingali, V. A Recursive Network Architecture. Tech. Rep. 626, USC/ISI, 2006.

356 Troxel, G., Blossom, E., Boswell, S., Caro, A., Castineyra, I., Colvin, A., Dreier, T., Evans, J.B., Goffee, N., Haigh, K., Hussain, T., Kawadia, V., Lapsley, D., Livadas, C., Medina, A., Mikkelson, J., Minden, G.J., Morris, R., Partridge, C., Raghunathan, V., Ramanathan, R., Santivanez, C., Schmid, T., Sumorok, D., Srivastava, M., Vincent, R.S., Wiggins, D., Wyglinski, A.M., Zahedi, S. Adaptive Dynamic Radio Open-source Intelligent Team (ADROIT): Cognitively-controlled Collaboration among SDR Nodes. In Networking Technologies for Software Defined Radio Networks, 2006. SDR '06.1st IEEE Workshop on, pp. 8–17 (2006).

357 Troxel, G., Caro, A., Castineyra, I., Goffee, N., Haigh, K., Hussain, T., Kawadia, V., Rubel, P., Wiggins, D. Cognitive Adaptation for Teams in ADROIT. In Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE, pp. 4868–4872 (2007).

358 Tschudin, C. Flexible protocol stacks. In SIGCOMM '91: Proceedings of the conference on Communications architecture & protocols, pp. 197–205, (ACM, New York, NY, USA1991).

359 Tschudin, C., Gold, R. Network pointers. SIGCOMM Comput. Commun. Rev., 33 (1):pp. 23–28, 2003.

360 Tschudin, C.F. Fraglets - a Metabolistic Execution Model for Communication Protocols. In 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park (2003).

361 Tsugawa, T., Hasegawa, G., Murata, M. Implementation and evaluation of an inline network measurement algorithm and its application to TCP-based service. In End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on, pp. 34 – 41 (2006).

362 Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the Lon-

don mathematical society, 42 (2):pp. 230–265, 1936.

363 Tyrrell, A., Auer, G., Bettstetter, C. Fireflies as role models for synchronization in ad hoc networks. In Proceedings of the 1st international conference on Bio inspired models of network, information and computing systems, BIONETICS '06, (ACM, New York, NY, USA2006).

364 Vandermeulen, F., Steegmans, F., Vermeulen, B., Vermeulen, S. Dynamically configurable protocol stacks. In Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on, vol. 3, pp. 1391–1394 vol.3 (2000).

365 Varghese, G. Network Algorithmics, (Chapman & Hall/CRC2010).

366 Vellala, M., Wang, A., Rouskas, G.N., Dutta, R., Baldine, I., Stevenson, D. A composition algorithm for the SILO cross-layer optimization service architecture. In Proceedings of the First International Conference on Advanced Network and Telecommunications Systems (ANTS) (2007).

367 Völker, L., Martin, D., El Khayaut, I., Werle, C., Zitterbart, M. A Node Architecture for 1000 Future Networks. In Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on, pp. 1–5 (2009).

368 Völker, L., Martin, D., Werle, C., Zitterbart, M., El Khayaut, I. Selecting Concurrent Network Architectures at Runtime. In Communications, 2009. ICC '09. IEEE International Conference on, pp. 1–5 (2009).

369 Wakamiya, N., Murata, M. Synchronization-based data gathering scheme for sensor networks. IEICE Transactions on Communications, 88 (3):pp. 873–881, 2005.

370 Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R. Utility functions in autonomic systems. In Autonomic Computing, 2004. Proceedings. International Conference on, pp. 70–77, (IEEE2004).

371 Walton, S., Hassan, O., Morgan, K., Brown, M. Modified cuckoo search: A new gradient free optimisation algorithm. Chaos, Solitons & Fractals, 44 (9):pp. 710 – 718, 2011.

372 Wang, Q., Zheng, H. Route and spectrum selection in dynamic spectrum networks. In Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE, vol. 1, pp. 625–629 (2006).

373 Watkins, C.J., Dayan, P. Q-learning. Machine learning, 8 (3-4):pp. 279–292, 1992.

374 Wetherall, D., Guttag, J.V., Tennenhouse, D. ANTS: a toolkit for building and dynamically deploying network protocols. In Open Architectures and Network Programming, 1998 IEEE, pp. 117–129 (1998).

375 Whitley, D., Rana, S., Heckendorn, R. The Island Model Genetic Algorithm: On separability, population size and convergence. CIT. Journal of computing and information technology, 7 (1):pp. 33–47, 1999.

376 Wiener, N. Cybernetics, (Technology Press1949).

377 Williamson, C., Wu, Q. A case for context-aware TCP/IP. SIGMETRICS Perform. Eval. Rev., 29 (4):pp. 11–23, 2002.

378 Winter, G., Galvan, B., Alonso, S., Gonzalez, B., Jimenez, J., Greiner, D. A Flexible Evolutionary Agent: cooperation and competition among real-coded evolutionary operators. Soft Computing, 9:pp. 299–323, 2005.

379 Wolf, T. Service-Centric End-to-End Abstractions in Next-Generation Networks. In Computer Communications and Networks, 2006. ICCCN 2006. Proceedings.15th International Conference on, pp. 79–86 (2006).

380 Wolf, T., Griffioen, J., Calvert, K.L., Dutta, R., Rouskas, G.N., Baldine, I., Nagurney, A. Choice as a principle in network architecture. SIGCOMM Comput. Commun. Rev., 42 (4):pp. 105–106, 2012.

381 Wolpert, D.H., Macready, W.G. No free lunch theorems for optimization. Evolutionary Computation, IEEE Transactions on, 1 (1):pp. 67–82, 1997.

382 Wood, D., Bruner, J.S., Ross, G. The Role of Tutoring in Problem Solving. Journal of Child Psychology and Psychiatry, 17 (2):pp. 89–100, 1976.

383 Wu, R., Chien, A., Hiltunen, M., Schlichting, R., Sen, S. A high performance configurable transport protocol for grid computing. In Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on, vol. 2, pp. 1117–1125 Vol. 2 (2005).

384 Xpressive Internet Architecture. http://www.cs.cmu.edu/~xia/.

385 Xiang, F., Junzhou, L., Jieyi, W., Guanqun, G. QoS routing based on genetic algorithm. Computer Communications, 22 (15–16):pp. 1392 – 1399, 1999.

386 Xu, R., Wunsch, D., I. Survey of clustering algorithms. Neural Networks, IEEE Transactions on, 16 (3):pp. 645–678, 2005.

387 Yamamoto, L., Tschudin, C. Experiments on the Automatic Evolution of Protocols Using Genetic Programming. In I. Stavrakakis, M. Smirnov (eds.), Autonomic Communication, vol. 3854 of Lecture Notes in Computer Science, pp. 13–28, (Springer Berlin Heidelberg2006).

388 Yang, X.S., Deb, S. Cuckoo search via Lévy flights. In Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on, pp. 210–214, (IEEE2009).

389 Yokoo, M., Hirayama, K. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In Proceedings of the Second International Conference on Multi-Agent Systems, pp. 401–408 (1996).

390 Zhang, W., Xing, Z. Distributed breakout vs. distributed stochastic: A comparative evaluation on scan scheduling. In AAMAS-02 Third International Workshop on Distributed Constraint Reasoning, pp. 192–201 (2002).

391 Zhao, J., Zheng, H., Yang, G.H. Distributed coordination in dynamic spectrum allocation networks. In New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on, pp. 259–268 (2005).

392 Zhao, Q., Xu, J., Liu, Z. Design of a novel statistics counter architecture with optimal space and time efficiency. SIGMETRICS Perform. Eval. Rev., 34 (1):pp. 323–334, 2006.

393 Zhou, C., Chia, L.T., Lee, B.S. Semantics in service discovery and QoS measurement. IT Professional, 7 (2):pp. 29–34, 2005.

394 Zitterbart, M., Stiller, B., Tantawy, A. A model for flexible high-performance communication subsystems. Selected Areas in Communications, IEEE Journal on, 11 (4):pp. 507–518, 1993.

395 Zitzler, E., Deb, K., Thiele, L. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. Evolutionary Computation, 8:pp. 173–195, 2000.

396 Zweig, J., Partridge, C. TCP alternate checksum options. RFC 1146 (Historic), 1990. URL http://www.ietf.org/rfc/rfc1146.txt. Obsoleted by RFC 6247.