Reliable Distributed Data Stream Management in Mobile Environments ☆

Gert Brettlecker[a,1,*], Heiko Schuldt[b]

[a]*Ergon Informatik*
Zürich, Switzerland
[b]*Databases and Information Systems Group*
University of Basel, Switzerland

Abstract

The proliferation of sensor technology, especially in the context of embedded systems, has brought forward novel types of applications that make use of streams of continuously generated sensor data. Many applications like telemonitoring in healthcare or roadside traffic monitoring and control particularly require data stream management (DSM) to be provided in a distributed, yet reliable way. This is even more important when DSM applications are deployed in a failure-prone distributed setting including resource-limited mobile devices, for instance in applications which aim at remotely monitoring mobile patients. In this paper, we introduce a model for distributed and reliable DSM. The contribution of this paper is threefold. First, in analogy to the SQL isolation levels, we define levels of reliability and describe necessary consistency constraints for distributed DSM that specify the tolerated loss, delay, or re-ordering of data stream elements, respectively. Second, we use this model to design and analyze an algorithm for reliable distributed DSM, namely efficient coordinated operator checkpointing (ECOC). We show that ECOC provides lossless and delay-limited reliable data stream management and thus can be used in critical application domains such as healthcare, where the loss of data stream elements can not be tolerated. Third, we present detailed performance evaluations of the ECOC algorithm running on mobile, resource-limited devices. In particular, we can show that ECOC provides a high level of reliability while, at the same time, featuring good performance characteristics with moderate resource consumption.

☆This work has partly been supported by the Swiss State Secretariat for Education and Research (SER) under contract No. SBF 03.0546-3 within the EU FP6 DELOS Network of Excellence on Digital Libraries.

*Corresponding author

Email addresses: `brettlecker@gmx.at` (Gert Brettlecker), `heiko.schuldt@unibas.ch` (Heiko Schuldt)

[1]The work has been done while the first author was with the Databases and Information Systems Group at the University of Basel, Switzerland.

Preprint submitted to Elsevier *October 15, 2010*

1. Introduction

In recent years, the proliferation of pervasive computing, wireless communication and sensor technology has spawned a variety of new applications in the area of *Data Stream Management* (DSM). In general, these applications are continuously monitoring the physical world by means of different sensors to extract and derive relevant information from multiple streams of data generated by these sensors. For example in healthcare applications, the continuous monitoring of patients at home (telemonitoring) is becoming more and more important, mainly due to the progression of chronic ailments in an aging society. A vital requirement of telemonitoring applications is that the DSM provides a high degree of reliability and availability, since it can potentially be life-saving. Another example is road traffic management, where the tremendous increase of vehicles requires the adoption of new traffic management systems to cope with limited road capacities. Also in traffic management, reliability of applications is an important aspect.

Consider, as an example, the following scenario: Fred, aged 68 and retiree, lives alone at his home. In the EU, 23 million adults are suffering from diabetes [24]. The diabetes disease affected Fred's heart, which has developed congestive heart failure (CHF). Fred is suffering from shortage of breath, swelling of the legs and ankles, pulse irregularity and palpitations, and difficulty with eating or sleeping. Due to his age, Fred also shows slight signs of dementia, which unfortunately affects the effectiveness of his personal disease treatment as he for instance forgets to take his medication. Without an assistive telemonitoring system, Fred has to do manual random sampling of his blood pressure, blood glucose level, heart rate, and body weight. For further examination, he has to consult his family doctor frequently. Nevertheless, this manual treatment does not prevent Fred from regular hospitalization due to dramatic degradations of his health state. In hospital, Fred's cardiac balance is restored by proper medication. Unfortunately, this balance is very unstable and hard to maintain by manual random sampling of physiological signs. As a vision for the future, Fred's caregiver decides to equip him with a wearable health monitoring systems consisting of a smart shirt [26], a ring sensor [2], a glucose measuring watch, and a PDA for local processing, intermediate storage, and wireless communication. This wearable setup will allow for unobtrusive monitoring of ECG, heart rate, respiratory and sweating rates, blood pressure, blood glucose level, blood oxygen saturation as well as motion activities, sensed with an inbuilt accelerometer. Fred's PDA will wirelessly communicate with the base station of his smart home system in order to extract and forward relevant streaming data to the caregiver. Besides that, Fred's smart home infrastructure also aggregates additional context measurements. For this reason, Fred's physical activity is detected by acceleration sensors attached to Fred's body and an integrated positioning system in the smart home environment. Additionally, an electronic scale is measuring body weight and fat and an electronic medication dispenser controls his medication. For this reason, the telemonitoring system builds up a distributed environment of nodes which is rather sensitive to numerous failure situations caused by unreliable mobile devices, sensors, and wireless communication. In order to achieve reliability, existing redundancy of devices, sensors, and communication channels has to be utilized by the smart home infrastructure.

In many practical settings such as the one presented above, data stream management is considered in a distributed environment and involves a large variety of mobile and embedded devices, like a doctor's PDA together with a patient's smartphone in the health telemonitoring scenario or different roadside sensor units in the traffic management scenario – together with stationary servers in healthcare institutions and traffic control centers, respectively. DSM applications which make use of mobile devices have to meet two important requirements. Firstly, they have to deal with limited resources, like CPU, memory, energy, or network bandwidth. This implies also constraints on applicable reliability strategies. For example, hot-standby strategies [18] require data processing to be done in parallel at multiple devices in order to achieve reliability. These strategies are in general too resource demanding, especially in environments where data streams are processed on resource-limited, mobile devices. Due to limited resources, individual operators on a device might fail without the complete device (node) to fail. Thus, failure handling needs to be considered at operator level, allowing to individually migrate failed operators to other nodes. Also, limited resources on a device may prevent that all operators of a failed node can be migrated but just of subset of them – so basically, all operators of a failed node might have to be distributed across different devices. Secondly, exploiting mobile devices for DSM applications also implies a highly increased failure probability compared to distributed computing scenarios involving only administered server computers and Ethernet connections. It is rather likely that a roadside sensor gets damaged due to an accident or even caused by animals or a wireless connection gets temporarily hampered by interference. Thus, making DSM fault-tolerant is a primary concern due to the high failure probability in mobile environments. In addition, the provable guarantees a DSM system is able to provide to its users are essential for applications like telemonitoring. There is a variety of research in the field of DSM but only a few groups have focused on reliable data stream management [19, 5, 4, 28, 14, 12, 10]. In particular, there exists no formal data stream model which allows to reason about the degree of reliability that can be guaranteed.

The contribution of this paper is threefold. First, we present a formal model of reliability in distributed DSM and apply this formalism to analyze reliability techniques for DSM [10, 12, 11]. In particular, we identify several levels which specify reliability and correctness of stream processing and which take into account the tolerated loss, delay, or re-ordering of data stream elements, similar to the way the SQL isolation levels specify the guarantees for concurrent transactions. Second, we apply this model to design and analyze the efficient coordinated operator checkpointing (ECOC) algoritm which provides a high degree of reliability for DSM with respect to constraints imposed by mobile devices. Third, we present an in-depth experimental evaluation on the performance of this reliability technique when applied in a mobile environment. In particular, we show that by properly designing algorithms for reliably handling data streams, a high degree of reliability is affordable in terms of the additional CPU and network overhead, even for mobile devices.

The paper is organized as follows: The basic data stream model is described in Section 2. Section 3 presents the formal failure model and reliability levels of DSM. Section 4 introduces and analyzes the ECOC algorithm for reliable DSM. Section 5 presents the experimental evaluation performed with our prototype DSM infrastructure

implementation OSIRIS-SE in a mobile environment. Section 6 surveys related work and Section 7 concludes.

2. Data Stream Model

The *DSM system* (DSMS) coordinates the execution *stream processes* on top of data stream operators in a network of loosely coupled *hosts* and ensures correctness even in case of failures. Table 1 gives an overview of symbols used in the remainder of this work.

2.1. Basic Data Stream Model

Let H denote the finite set of all hosts participating within the distributed DSM system. Each host is able to perform certain data stream operations, also called *operator types*. The finite set of all operator types available for execution within the DSM system is called OT. The subset of all operator types available at a given host $h \in H$ is called $OT(h)$.

Stream process definitions combine operator types to build up complex stream processing tasks which consume and produce data streams of the *outside world* which encompasses all external systems interacting with the DSMS. The DSMS executes instances of stream process definitions, called *stream processes*. Stream processes make use of instances of operator types, also called *operators*. Operators continuously process *data streams*. In the following, we introduce these terms with formal definitions.

A *DSM system* (DSMS) is defined as the following 4-tuple:

$$\mathcal{DSMS} = \langle H, OT, SPD, SP \rangle$$

where H is a finite set of hosts participating in the DSM system, OT the finite set of globally available operator types offered by all hosts, SPD is a set of available stream process definitions, and SP is the set of running stream processes. □

A *data stream* (DS) is a possibly infinite, totally ordered set of data stream elements $(DE, \prec, op, ip, \Sigma)$. DE is the set of all elements de within a stream. The connection point of a data stream at producer-side is called *output-port* op and at consumer-side *input-port* ip, respectively. □

Input and output naming is assigned from a consumer/producer point of view. The symbols to be sent as payload within data stream elements are defined in the *data stream alphabet* Σ. A DS represents a continuous transmission of data stream elements de in a temporal order between a producer and a consumer. The notation $DS.ip$ refers to the input-port of the data stream DS and $DS.op$ refers to the output-port, respectively.

A *data stream element* $de \in DS$ is defined by the following tuple: $de = \langle \tau, \xi, pd \rangle$ where $\tau \in \mathbb{R}^+$ is a global timestamp attribute which is given to a data stream element at the time of processing, $\xi \in \mathbb{N}_0$ is a sequence number which is used for ordering and gap detection of the data stream elements within the data stream, and pd is the payload information, which is a symbol of the data stream alphabet $pd \in \Sigma$. □

Although a data stream element shows some similarities to a tuple of a relational database table, especially since both are structured according to a given schema, the

number of data stream elements in a data stream is potentially unlimited. In order to ease the notation of structured objects and their attributes, we use a labeled notation. For example, the payload attribute pd of a data stream element de is denoted as $de.pd$.

According to the previous definition data stream elements have two temporal ordering contexts. First, a global timestamp, called *processing time* and second, a logical sequence number, called *stream time*. The global timestamp specifies in absolute terms *when* a particular stream element has been processed. Thus, it allows for quantitative measurements of processing and transmission delays. In contrast, the stream time specifies in relative terms the *order* of data stream elements. Hence, the stream time allows for detection of gaps and disorder in data stream elements.

For example, a data stream element coming from a sensor and arriving at a processing operator may look like the following:

$de_{189} = \langle$
    29.02.2008 10:00:00.50,
    189,
    $< sample >$
        $< time >$ 29.02.2008 $10:00:00.00$ $< /time >$
        $< value > 10mV < /value >$
    $< /sample >$
$\rangle$

The data stream element is the 189th element within the data stream (stream time). It was received by the operator at 10:00:00.50 on 29.02.2008 global time (processing time). The payload of the element contains a structured entry, e.g., in XML format. The XML format is displayed here for illustrative reasons. In practice, a more compact data representation may be chosen. The payload was generated by the sensor and contains the global time of generation and the value 10mV (e.g., a sensor voltage value that corresponds to a physical quantity). Moreover, this example illustrates that processing time and stream time give no information about the original time the data stream element was generated by the sensor (*generation time*). If this information is necessary for the application it must be part of the payload information (as it is the case in the example given above). Of course, if data streams coming from different sensors are joined on the basis of generation time, the clocks of the different sensors have to be synchronized.

Based on the data stream definition, we define *operator types* which are in charge of processing data stream elements of incoming data streams and producing derived data stream elements of outgoing data streams. The connection points of data streams to the operator type are called *ports*. Each operator type ot has an ordered set of $n \in \mathbb{N}_0$ input-ports IP_{ot} with cardinality $|IP_{ot}| = n$ and an ordered set of output-ports OP_{ot} with cardinality $|OP_{ot}| = m$. Each output-port $op_i \in OP_{ot}$ produces data stream elements with the stream alphabet $op_i.\Sigma$. Accordingly, each input-port $ip_i \in IP_{ot}$ expects to receive data stream elements with the stream alphabet $ip_i.\Sigma$. The behavior of an *operator type* is deterministic and can be modeled as finite state machine (FSM). Figure 1 illustrates the operator type model. This means that we assume an operator type to always produce output stream elements with the same payload for the same input stream elements (note that global timestamp and stream time are part of the input stream ele-

ments). However, in a real-world setting, processing time and transmission delays may have an impact on the output data streams (e.g., on their stream time and or processing time). For example, an operator type implementation running on a highly loaded device may induce a processing delay that leads to a timeout. Hence, processing results, i.e., the results produced by an operator type implementation in a particular system environment, can differ from the ideal case and are not supposed to be deterministic.

Two streams shown in Figure 1 have a special purpose. First, the *config stream* allows for explicitly changing the operator state. This can be used, for instance, for operator initialization. A sample operator, in XML representation, processing medical alerts may have the following internal state:

$< State >$
$\quad < BeginOfMeasurement >$
$\qquad$ 29.02.2008 10:00:00
$\quad < /BeginOfMeasurement >$
$\quad < AlertsOverall >$
$\qquad$ 16
$\quad < /AlertsOverall >$
$\quad < AlertsLastHour >$
$\qquad$ 1
$\quad < /AlertsLastHour >$
$\quad < StartOfCurrentHour >$
$\qquad$ 30.10.2007 18:00:00
$\quad < /StartOfCurrentHour >$
$< /state >$

The operator counts the number of medical alerts since the beginning of the measurement and the number of medical alerts during the last hour.

Second, the *state backup stream* allows for reading the current operator state. This allows to take checkpoints of an operator A during execution, if necessary, in order to deal with failures and to achieve a high degree of reliability by migrating A's state to another operator B. Details on operator checkpointing are presented in Section 4.

An *operator type* OT is defined by the following tuple:

$$OT = \langle \Theta, \Gamma, ST, \delta, \omega, IP, OP, min \triangle \tau \rangle$$

where Θ is the input alphabet as cartesian product over the alphabets of all n data streams received by all input-ports IP:

$$\Theta = \prod_{i=0..(n-1)} (ip_i.\Sigma \cup \{nd\})$$

and where Γ is the output alphabet as cartesian product over the alphabets of all m data streams produced by the ordered set of output-ports OP:

$$\Gamma = \prod_{i=0..(m-1)} (op_i.\Sigma \cup \{nd\})$$

ST is a finite, non empty set of operator states. The state transition function δ is defined as: $\delta : ST \times \Theta \rightarrow ST$ and the output function ω is defined as: $\omega : ST \times \Theta \rightarrow \Gamma$.

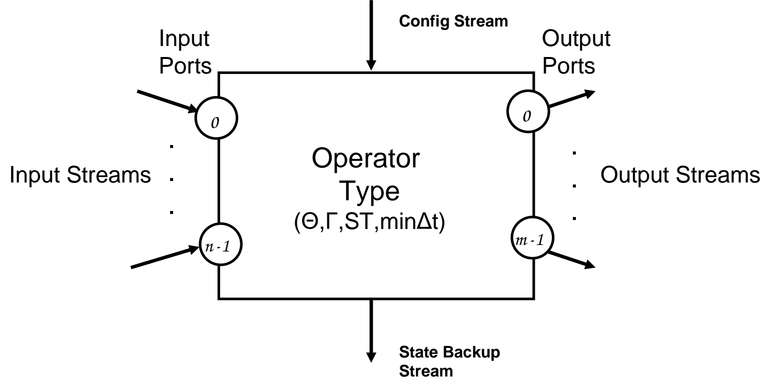


Figure 1: Operator Type Model

Finally, $min \triangle \tau \in I\!R^+$ describes a minimal delay for processing a single state change of this operator type. □

An operator type describes a deterministic finite state machine. Whenever new input data stream elements are available, δ is applied in order to proceed to the new state and ω is applied in order to produce output data stream elements. Of course, the processing delay of a state change $\triangle\ \tau$ of an operator type instantiation in real-world depends on various factors, e.g., speed and architecture of CPU. An operator type defines an upper limit $min \triangle \tau$ for this minimal processing delay that a real world instantiation of the operator type has to guarantee. Hence $min \triangle \tau$ does not specify the actual processing delay of a concrete operator instance.

The nd symbol identifies input or output data streams where no data stream element is present. In this model, the FSM works asynchronously which means that the FSM has not to wait for data stream elements to be present at each input in order to proceed to the next state. Similarly, not every state change has to produce data stream elements at each output-port.

Based on the two basic DSM building blocks of operator types and data streams, we are now able to combine these two in order to define complex stream processing tasks, called *stream process definitions*. A stream process definition is described as a directed multigraph, where the vertices are operator types and the edges are data streams. The graph is a multigraph because more than one edge (data stream) can connect a pair of vertices (operator types). For example, we consider a combined heart and blood pressure sensor, which generates a heart activity and a blood pressure data stream as two distinct output data streams and an analysis operator type which combines blood pressure and heart activity as two distinct input data streams. In this case, we have two edges (data streams) from the combined sensor to the analysis operator type.

A *stream process definition*, $spd = \langle V, E \rangle$ describes a directed multigraph, where V is a finite set of vertices and E is a finite set of edges. The set of vertices V of spd is

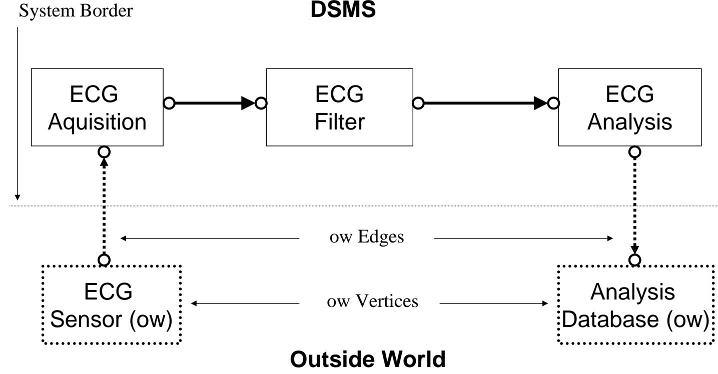


Figure 2: Example Stream Process Definition

defined as:

$$V = \{\langle ot, s_o, \triangle\tau\rangle \,| ot \in OT \cup \{ow\}, s_0 \in ST, \triangle\tau \in \mathbb{R}^+\}$$

where ot is describing the operator type of the vertex. In addition to the operator types offered by the DSMS, outside world interactions are represented by ow-vertices. The initial state of the vertex using the operator type is given by s_0, this state is set by sending along the *config stream* of the operator instance at the time of startup. In order to specify a maximum tolerable delay for processing a state transition at this vertex, $\triangle\tau$ is given. □

The set of edges E of spd is defined as:

$$E = \{\langle x, op, y, ip, \triangle\tau\rangle \,|$$

$$x \in V, op \in x.ot.OP, y \in V, ip \in y.ot.IP, \triangle\tau \in \mathbb{R}^+\}$$

where x is the source vertex, op is the corresponding output port at the operator type ot of vertex x. Similarly, y is the destination vertex, ip is the corresponding input port at the operator type ot of vertex y. Edges where either the source vertex or the destination vertex is an ow-vertex are called ow-edges. $\triangle\tau$ specifies a maximum tolerable transfer delay for a data stream element along this edge. □

Figure 2 illustrates a simple example stream process definition taken from a telemonitoring application by applying the presented DSM model. The stream process offers heart activity analysis by applying three operators. The first operator reads out the sensor device. The second operator applies filtering at signal level in order to remove noise. Finally, the third operator derives medically relevant information of the *electrocardiogram (ECG)*. Consequently, the sample stream process consists of three vertices where $ot \in OT$ and two outside world vertices where $ot = ow$. The sensor device is an ow-vertex. The acquisition is already part of the DSMS. Furthermore, filtering and analysis of the heart signal is performed. Finally, the DSMS is producing a data stream feeding an analysis database, which is an ow-vertex again.

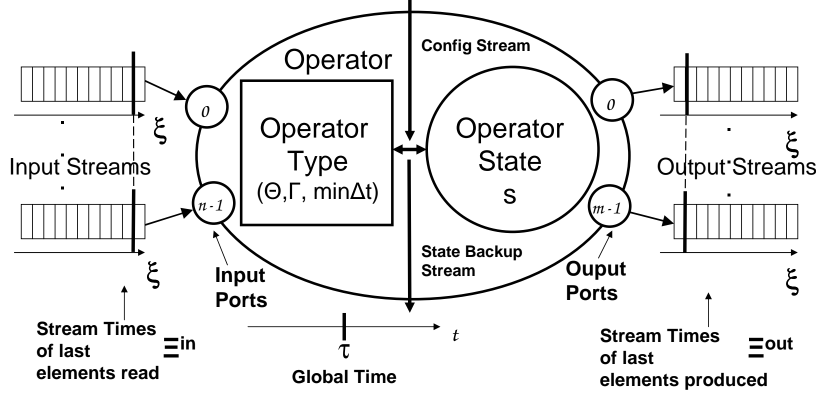


Figure 3: Operator Model

2.2. Stream Process Execution

When a running instance of an operator type is generated at a host during the execution of a stream process, this instance is called *operator*. In addition to the operator type, the operator has a current time context, a current state and a current host. Figure 3 illustrates the operator model.

An *operator* o is a running instance of a vertex of a stream process and has the following definition:

$$o = \left\langle v, \Xi^{in}, \Xi^{out}, s, h, \tau \right\rangle$$

where $v \in spd.V$ is a vertex taken from the corresponding spd, Ξ^{in} is the set of stream times of the last data elements de processed at each input port. Analogously, Ξ^{out} is the set of stream times of the last de processed at each output port. The current state of the operator is given by $s \in ST$, the host executing the operator instance is $h \in H$, and the current global time is indicated by τ. □

In order to initialize running operator instances, a data stream element with a new initial state as payload can be sent to the operator instance along the config stream. In order to retrieve the current state of an operator during execution, the state backup stream produces a data stream element with the current state as payload information for each state transition of the operator.

Finally, running operator instances form together a running instance of a stream process definition, called *stream process*. In addition to the stream process definition the stream process has a set of operator instances. In this set of operators, we do not consider outside world ow-vertices because these operator instances are not executed within the modeled DSM system (DSMS).

A *stream process*, $sp = \langle spd, O, DSS \rangle$ has a corresponding stream process definition spd and a set of operators O executing the operator types given by spd. DSS is a set of data streams connecting the operator instances. □

2.3. Well-Formed Stream Processes

In this section, we describe the set of constraints a stream process definition has to fulfill in order to be considered valid or *well-formed*. In a practical system, these constraints should already be evaluated by the stream process design tools before process execution.

1. Each vertex in a stream process definition has *exactly one* input edge attached to each input port.
2. Each vertex in a stream process definition has *at least one* output edge attached to each output port.
3. The stream alphabet of each output port corresponds along all connected edges to the expected stream alphabet of input ports.
4. Outside world ow-edges are always connecting ow-vertices with non ow-vertices. The stream process definition is not describing interactions between ow-vertices.
5. Reasonable delay constraints are given within the stream process definition, which means the $min \bigtriangleup \tau$ constraints given by the operator type are not violated in the spd.

More formally, a *well-formed stream process* (spd) definition holds the following constraints:

$$\forall v \in spd.V (\forall ip \in v.ot.IP (\exists! e \in spd.E : e.ip = ip)) \;\; (1)$$

$$\forall v \in spd.V (\forall op \in v.ot.OP (\exists e \in spd.E : e.op = op)) \;\; (2)$$

$$\forall e \in spd.E : e.op.\Sigma = e.ip.\Sigma \;\; (3)$$

$$\nexists e \in spd.E : (e.x = ow \wedge e.y = ow) \;\; (4)$$

$$\forall v \in spd.V : v. \bigtriangleup \tau \geq v.ot.min \bigtriangleup \tau \;\; (5)$$

□

Finally, well-formed stream process definitions may contain cycles in the stream process definition multigraph. Cycles are paths within the stream process definition, where the start vertex corresponds to the end vertex. We distinguish between cyclic and non-cyclic stream process definitions.

2.4. Well-Activated Stream Process

After the design of a well-formed stream process definition spd, the DSMS is in charge of activating a stream process instance sp of the given spd. After successful activation, the following constraints on *well-activated* stream process instances are defined:

1. Each non ow-vertex in spd has a corresponding running operator instance in O.
2. Each edge in spd has a corresponding running data stream instance in DSS.

More formally, a *well-activated stream process* (sp) holds the following constraints:

$$\forall v \in sp.spd.V|_{v \neq ow} (\exists! o \in sp.O : v = o.v) \;\; (1)$$

$$\forall e \in sp.spd.E (\exists! ds \in sp.DSS :$$
$$(e.op = ds.op \wedge e.ip = ds.ip)) \;\; (2)$$

□

2.5. Reliability Levels of DSM

The presented model describes all interactions of the DSMS with the outside world through outside data streams (ow-edges). Therefore, from the outside world point of view, the internals of DSMS can be considered a black box. Based on this fact, we can define the reliability level of a DSMS as seen by an outside user by comparing a *real-world DSMS* to an *ideal DSMS*. The *ideal DSMS* system is a virtual system which executes all stream processes in a proper way according to the stream process definition (spd). This means that no delays happen during stream process execution (i.e., no processing time needed by the operator instances and also no time needed for transferring data stream elements between operator instances), no data stream elements get lost, and no misordered data stream elements are produced due to failures. In contrast, the *real-world DSMS* is an error-prone DSMS system that has to deal with failures happening in the real-world.

In the following, we compare the *output outside data streams* DS_i of an ideal DSMS with output outside data streams DS_r of a real-world DSMS. Since the input outside data streams are generated by the outside world, which is per definition failure-free, we only need to compare output outside data streams.

The highest level of reliability has a real world system with identical output data streams as the ideal system. In this case, there is no difference to the ideal DSMS from the outside world's point of view.

The *ideal reliability level* is achieved when for all output data streams $DS_i = DS_r$ holds. □

Unfortunately, real world systems have to cope with failures. Therefore, in general, they do not produce the same result as ideal systems. For this reason, we define the following two subset data streams of a real world output outside data stream.

The *correct data stream* DS_c contains the subset of the data stream elements appearing in the real-world data stream DS_r (i.e., $DS_c \subseteq DS_r$) which have the same (correct) sequence number ξ and payload information pd as data stream elements appearing in the corresponding ideal data stream DS_i:

$$\forall de_c \in DS_c(\exists! de_i \in DS_i : (de_i.\xi = de_c.\xi \land de_i.pd = de_c.pd))$$

□

The *incorrect data stream* DS_f contains the subset of the data stream elements appearing in the real-world data stream DS_r (i.e., $DS_f \subset DS_r$) which have no corresponding data stream elements appearing in the ideal data stream DS_i with respect to sequence number ξ and payload information pd:

$$\forall de_f \in DS_f(\nexists de_i \in DS_i : (de_i.\xi = de_f.\xi \land de_i.pd = de_f.pd))$$

□

Furthermore, we are able to degrade the ideal reliability level in three orthogonal dimensions considering *loss, delay,* and *order*, respectively.

Loss is defined as having a smaller number of correct data stream elements DS_c in the real-world output outside data stream than the ideal output outside data stream, but may have additional incorrect data stream elements DS_f. The loss definition does

not put constraints on the global timestamp τ, hence loss is neither related to delay nor order.

Limited-loss reliability level (LILO) is defined as having:

$$DS_r = DS_c \cup DS_f$$
$$|DS_i| \geq |DS_c| \geq |DS_i| * LF \quad LF \in (0, 1]$$
$$|DS_f| \leq |DS_i| * EF \quad EF \in [0, \infty)$$

where LF is a maximum allowed loss factor and EF is a maximum allowed error factor. □

Lossless reliability level (LOLE) is a special case of the loss reliability level, where $LF = 1$ and $EF = 0$. □

Delay is defined as having a certain delay in the global timestamp τ compared to DS_i for correct data stream elements in the real-world outside data stream. Delay is not putting any constraints on missing correct data stream elements, which are subject of loss. In general, delay is also not putting any constraints on the order of data stream elements within global time.

Limited-delay reliability level (LIDE) is defined as having:

$$\forall de_c \in DS_c (\exists! de_i \in DS_i :$$
$$de_i.\xi = de_c.\xi \wedge de_i.pd = de_c.pd$$
$$\wedge\ de_i.\tau \leq de_c.\tau \leq de_i.\tau + \triangle\tau)$$

where $\triangle\tau$ is a maximum allowed delay. □

Delay-free reliability level (DEFR) is a special case of the limited-delay reliability level, where $\triangle\tau = 0$. Particularly in the delay-free case, the order of the data stream elements is preserved. □

Order is defined for correct data stream elements in the real-world outside data stream and comes in two flavors. One is the *intra stream order*, which means that the ordering of the data stream (according to the global timestamp τ) is preserved.

Intra-stream order preserving reliability level (IASO) guarantees:

$$\forall de_c(\xi) \in DS_c : de_c(\xi).\tau \leq de_c(\xi + 1).\tau$$

□

The other is *inter-stream order*, were we additionally compare whether the temporal-order of data stream elements is preserved between data streams.

Inter-stream order preserving reliability level (IESO) guarantees:

$$\forall de_{c1} \in DS_{c1}, \forall de_{c2} \in DS_{c2}, \forall de_{i1} \in DS_{i1}, \forall de_{i2} \in DS_{i2} :$$
$$de_{i1}.\xi = de_{c1}.\xi \wedge de_{i1}.pd = de_{c1}.pd \wedge de_{i2}.\xi = de_{c2}.\xi \wedge de_{i2}.pd = de_{c2}.pd \wedge ($$
$$((de_{i1}.\tau \leq de_{i2}.\tau) \wedge (de_{c1}.\tau \leq de_{c2}.\tau))$$
$$\vee((de_{i1}.\tau > de_{i2}.\tau) \wedge (de_{c1}.\tau > de_{c2}.\tau)))$$

□

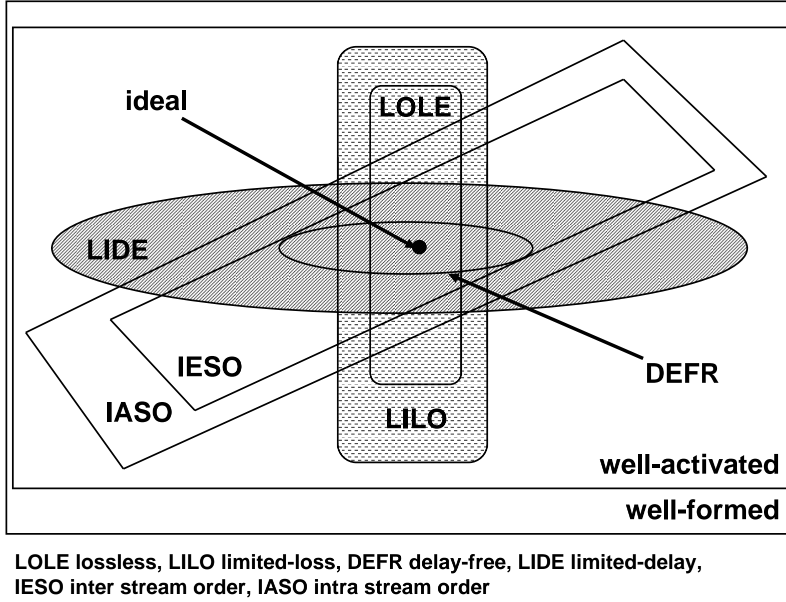


Figure 4: Reliability Levels of DSM

Figure 4 illustrates the reliability levels of DSM and their relationship. The three levels of reliability LILO, LIDE, IASO represent orthogonal sets. Lossless (LOLE), delay-free (DEFR), and inter stream order (IESO) are subsets of LILO, LIDE, and IASO respectively. The only exception of orthogonality is between DEFR and IESO/ IASO are related because delay-free reliability guarantees inter-stream and intra-stream order reliability but not vice versa. The formal proofs for the pairwise relationships between reliability levels can be found in [9].

3. Runtime Behavior of a DSMS

3.1. Failure Model

In the following, we describe a failure model based on the given DSMS model and analyze the consequences of failures at the operator, data stream and host levels, respectively. Note that this failure model encompasses the failures of *all* operator instances on a particular host, but also takes into account that *individual* operator instances at a host might fail, for instance due to an 'out-of-memory' exception in an overload situation, without impacting other instances at that host. The latter situation is likely to occur, for example, if operator instances are hosted on resource-limited (mobile) devices.

Failure of an operator instance o_f: This failure affects the stream process SP_f which is using the affected operator:

$$SP_f \in SP|_{o_f \in SP_f.O}$$

Symbol	Description
$DSMS$	*Data Stream Management System* including all operators and data streams
h	A host participating in the $DSMS$
H	Set of hosts
ot	A operator type available for instantiation in the $DSMS$
OT	Set of operator types
spd	A stream process definition available for instantiation in the $DSMS$
SPD	Set of stream process definitions
sp	A running stream process instance in the $DSMS$
SP	Set of stream process instances
DS	A data stream
DE	A data stream element
op	An output port, the producer end of a data stream
OP	Set of output ports
ip	An input port, the consumer end of a data stream
IP	Set of input ports
Σ	An alphabet of symbols appearing as payload of a data stream
τ	The current global time
ξ	The stream time associated with a data stream element
pd	The payload information of a data stream element
Θ	The cartesian product of all input data stream alphabets of an operator type
Γ	The cartesian product of all output data stream alphabets of an operator type
ST	Set of possible operator states of an operator type
s	Current state of a running operator instance
s_0	Initial state of a running operator instance
δ	State transition function of an operator type
ω	Output function of an operator type
nd	No data stream element present symbol
v, x, y	Vertex of a stream process definition
V	Set of all vertices in a stream process definition
e	Edge of a stream process definition
E	Set of all edges in a stream process definiton
ow	Vertex outside of the $DSMS$
o	Running operator instance
O	Set of running operator instances of a stream process instance
DSS	Set of data streams connecting operators of a stream process
$\triangle\tau$	Processing delay
Ξ^{in}	Set of stream times of the last processed input data stream elements
Ξ^{out}	Set of stream times of the last generated output data stream elements
DS_c	A correct data stream
DS_r	A real-world data stream
DS_f	A incorrect data stream
TC	Time context of a running operator instance
TS	Transfer state of a running operator instance
RI	Routing information of a running operator instance

Table 1: Symbol Description Table

Moreover, within SP_f besides o_f all data streams DS_f from DSS are affected that are connected to a port of o_f:

$$DS_f \in SP_f.DSS|_{(DS_f.ip \in o_f.v.ot.IP) \vee (DS_f.op \in o_f.v.ot.OP)}$$

Failure of a data stream DS_f: This failure affects the stream process SP_f which is using the affected data stream:

$$SP_f \in SP|_{DS_f \in SP_f.DSS}$$

Moreover, within SP_f, in addition to DS_f all operator instances o_f that are connected by DS_f are affected:

$$o_f \in SP_f.O|_{(DS_f.ip \in o_f.v.ot.IP) \vee (DS_f.op \in o_f.v.ot.OP))}$$

Failure of a host h_f: This failure affects all stream processes SP_f having running operator instances, which are hosted by the affected host:

$$SP_f \in SP|_{(\exists o_f \in SP_f.O: o_f.h = h_f)}$$

Within each affected stream process SP_f the following operators o_f are affected:

$$o_f \in SP_f.O|_{o_f.h = h_f}$$

Moreover, also the following data streams DS_F connecting to an affected operator o_f are affected by the failure:

$$DS_f \in SP_f.DSS|_{(DS_f.ip \in o_f.v.ot.IP) \vee (DS_f.op \in o_f.v.ot.OP)}$$

Our failure model assumes all failures to be *fail-stop failures*. Fail-stop failures are failures where the affected part is completely stopping its work. The outside world system is assumed to be always working correctly.

It should be noted that the approach presented in this paper jointly addresses different types of failures, including *failures at operator level*. This is in contrast to other work in the field, such as [4, 14], that only focuses on failure handling at host granularity. However, more fine-grained failure handling at operator level is beneficial for two main reasons. Firstly and most importantly, our approach is tailored to handle failures in mobile environments which are characterized by devices with limited resources. In such environments, operator failures may be triggered by overload situations. Typically, memory or CPU load limits are reached and some (but not necessarily all) running operator instances on a mobile host may suffer from memory allocation errors or thread starvation. This leads to the situation that one or more operators on a mobile host are subject to operator failures. Addressing failure handling at operator level allows to restrict operator migration to these failed operators while keeping the other operator instances that are working properly on the device. Secondly, the more fine-grained approach at operator level subsumes failure handling at node level. However, it allows to individually treat all local operators that fail due to a node failure. This means that they can be migrated independently of each other. Again, when considering devices with limited resources, the individual migration of the operators of a

failed node to different nodes can be highly beneficial in order to prevent new hosts to be overloaded after migration.

In our approach, we explicitly exclude software bugs as reason for operator failures since such bugs can not be managed by recovery mechanisms. In our model, we assume that all operator types have been verified correctly, i.e., they do not feature inherent logical failures.

When an operator fails, the subsequent operator will not receive any data stream elements because of the fail-stop assumption. Subsequent operators will wait for recovery but do not fail themselves. If there is a data stream failure between two connected operators we assume that the affected operators do not fail and will wait for recovery of the failure situation.

Moreover, our approach assumes the DSMS infrastructure offers a reliable FIFO-transport for data stream elements. This FIFO-transport already guarantees intra-order reliability of DSM as described in Section 2.5.

3.2. States Within a Stream Process

During runtime of a stream process in a DSMS different kinds of states are generated (illustrated in Figure 5):

- **Operator state** ($s(\tau)$). This is the most obvious state and has already been introduced in Section 2.1. This state is generated by each running operator instance during the processing of data streams.
- **Time context** ($TC(\tau)$). Each operator instance has to know its current stream-time context.
$$TC(\tau) = \left\langle \Xi^{in}, \Xi^{out} \right\rangle$$
where Ξ^{in}, Ξ^{out} refer to the stream-time of the last processed input and output data stream elements (c.f. operator definition in Section 2.1).
- **Transfer state** ($TS(\tau)$). In a real world system the processing and transmission of a data stream DS suffers from delays. Therefore, there may be at a given global timestamp τ some data stream elements in a state of transfer along an edge $e = \langle x.op, y.ip, \triangle\tau \rangle$ of a stream process. The state of the data stream between *op* and *ip* is called *transfer state* TS. The transfer state along an edge is given as subset of DS:
$$TS(\tau) = DS(\xi_s, \xi_e)$$
where $\xi_s \leq \xi_e$ and $|DS(\xi_s, \xi_e)| = \xi_e - \xi_s$ is the number of elements in the transfer state. ξ_s refers to the element with oldest stream-time and ξ_e refers to the element with the newest stream-time in state of transfer. In practice, an *acknowledgment* protocol between consumers and producer of a data stream will describe elements in transfer as *non-acknowledged* elements.
- **Routing information** ($RI(\tau)$). Finally, for each edge the stream process execution has to know the host which is currently hosting the source vertex x and the host of the destination vertex y, where the data stream has to be send to. This *routing information* h_x, h_y is also considered as state of the stream process.

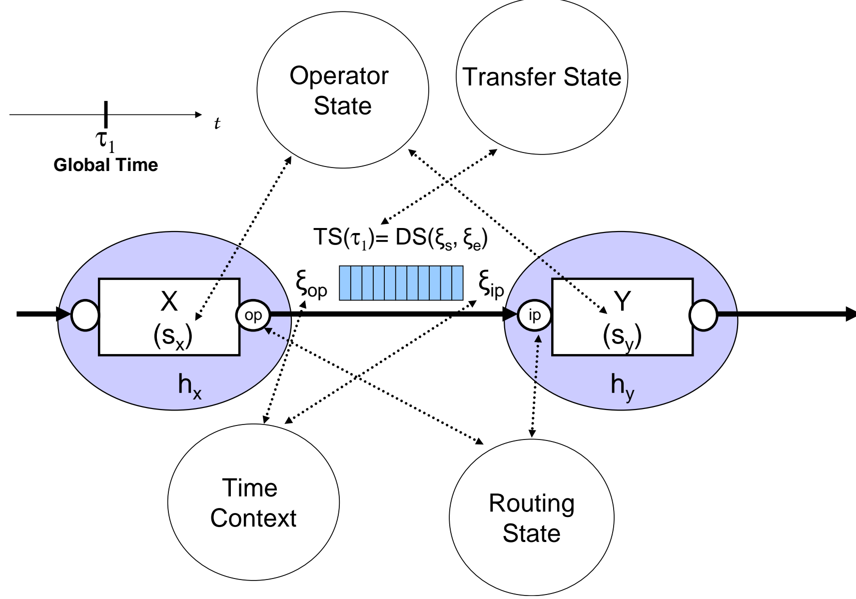


Figure 5: States of a Stream Process

3.3. Consistency Within a Stream Process

Based on the previous definitions of states within stream processes, we are able to define consistency constraints for relations between the *time context* of the *operator state* and the *transfer state*.

Exact consistency constraints ensure that at every point in time τ during processing, all operator instances are working consistently with regard to the transfer states of the data streams in between. This means that no operator instance has lost any data stream element or has processed data stream elements multiple times. Furthermore, all internal states of the operator instances are correct with regard to the processed data stream elements. Hence, all operator instances are consistent to each other. In this case, exact consistency guarantees the lossless reliability level, because loss would cause an inconsistency.

Exact Consistency is defined as follows: For all pairs of two operators x, y which are connected via an edge e from $x.op$ to $y.ip$ at any point in global time τ the following has to hold

$$\begin{aligned} x(\tau) &= \left\langle OT_x, \Xi_x^{in}, \Xi_x^{out}, s_x, h_x, \tau \right\rangle \\ y(\tau) &= \left\langle OT_y, \Xi_y^{in}, \Xi_y^{out}, s_y, h_y, \tau \right\rangle \\ TS(\tau) &= DS(\xi_s, \xi_e) \\ (\xi_{op} &= \xi_e) \wedge (\xi_{ip} = \xi_s) \end{aligned}$$

where ξ_{op} is the current stream time of the last element produced at the output port of

x taken from Ξ_x^{out} and ξ_{ip} is the current stream time of the last element consumed at the input port of y taken from Ξ_y^{in}. □

Based on the deterministic operator model, we are able to relax the exact consistency constraint. Relaxed consistency allows for the sender x to re-send data stream elements if necessary and for the receiver y to receive data stream elements again. An appropriate transport mechanism within the DSMS is able to guarantee these assumption for internal edges. For interaction with the outside world along ow-edges, these assumptions require to allow to re-read data stream elements coming from sensors and to re-send data stream elements to outside world receivers. Of course, the available time window within the data stream for re-reading and re-sending is limited by an acknowledgement mechanism.

Relaxed Consistency is defined as follows: For all pairs of two operators x, y which are connected via an edge e from $x.op$ to $y.ip$ at any point in global time τ:

$$x(\tau) = \left\langle OT_x, \Xi_x^{in}, \Xi_x^{out}, s_x, h_x, \tau \right\rangle$$
$$y(\tau) = \left\langle OT_y, \Xi_y^{in}, \Xi_y^{out}, s_y, h_y, \tau \right\rangle$$
$$TS(\tau) = DS(\xi_s, \xi_e)$$
$$(\xi_{op} \leq \xi_e) \wedge (\xi_{ip} \geq \xi_s)$$

□

Having relaxed consistency enforced at all times during the processing of a DSMS guarantees lossless reliability. Re-sending or re-reading data stream elements within our deterministic finite state machine model is not generating loss (which also includes wrong data stream elements) because the replayed elements are exact duplicates of the original elements and only applied if needed. Duplicate detection based sequence numbers allows to drop unnecessary duplicates without affecting the correctness of DSM processing. Please note that data stream elements both contain their global timestamp and their stream time. Hence, re-reading complete streams will not impact the output produced by this operator. However, it might have an impact on the output of a subsequent operator, as the processing time of the data stream elements received by this operator might be different.

3.4. Distinction Between Delays and Failures

In a real-world DSM system, the processing and transmission of data stream elements is commonly subject to delays. Our model has inherently accepted delays as part of operator types and stream process definitions where vertices and edges are associated with maximum allowed delay constraints. Given these delays, we can define *temporary failures*, where the effect of a failure is only temporary, for instance due to a wireless network disturbance. On the other hand, if a failure is persistent in a way that the delay constraints are exceeded, we consider the failure as *permanent failure*. In a real-world implementation of DSMS the temporary failures are usually compensated by having buffers between pairs of operators. Contrarily, permanent failures have be treated in a more sophisticated way. In order to guarantee the limited delay reliability level, the DSMS system has to resolve the failure situation before the maximum allowed delay $\triangle \tau$ is reached. Since the reliability strategy needs some time τ_r to recover

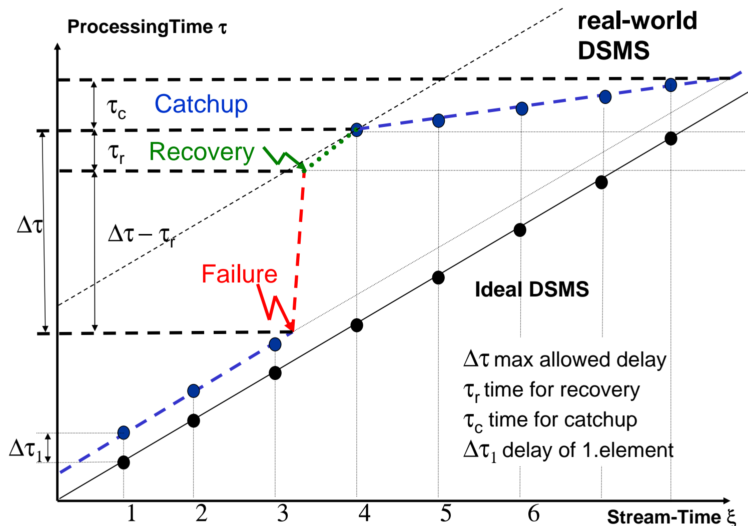


Figure 6: Temporal Behavior of a Failure

from the failure situation, the DSMS has to start failure handling when the current delay exceeds the maximum allowed delay $\triangle\tau$ minus the recovery time τ_r. Figure 6 illustrates the failure handling of a DSMS over stream-time (x-axis) and processing time (y-axis). After the failure has been resolved the DSMS needs additional *catchup time* τ_c to work-off the congestion caused during time of failure and reduce the delays back to average level.

4. Reliable Operator Execution

In this section, we formally analyze the degree of reliability provided by a family of algorithms addressing coordinated checkpointing [12] based on the data stream model presented in Sections 2 and 3. We focus in particular on guaranteeing relaxed consistency which results in lossless reliability.

4.1. Operator Migration

During runtime of a distributed DSMS, failures as described in Section 3.1 are likely to happen. If a failure situation persists longer than $\triangle\tau - \tau_r$, at any edge or vertex of a running stream process, the DSMS has to actively apply some mechanism in order to recover from the failure situation and to keep the required reliability level. The expected failure situations in our model are operator failures, data stream failures, and host failures. In the case of an operator failure, only an operator instance has failed and may be restarted at its current host. In the case of a data stream failure, a data stream connection between two operators has failed, e.g., due to a network disconnection.

In the case of a host failure, a host of the DSMS has failed and all running operator instances have to be migrated to other unaffected hosts. Different kinds of operator failures or data stream failures may be solved by restarting an operator at the same host or by re-establishing a data stream between two operators, maybe by using a different kind of network connection.

In this work, we focus on failure situations which can be resolved by migration of running operator instances from the affected hosts to other unaffected hosts. This movement of an operator instance from one host to another is called *operator migration*. Operator migration implies the redirection of data streams (which corresponds to the replacement of edges in the data stream graph).

Of course, there are failure situations that can not be handled by operator migration. For example, if a data stream failure persists and making communication between hosts executing operators of the current stream process impossible. These failure situations are out of scope of this work.

In our work, we do not focus on the detection of such failure situations in a DSMS. We assume that failures are detected and that there is a significant number of unaffected hosts in the network available to take over the workload.

For this reason, we consider a passive-standby approach based on checkpointing [15, 19]. We define an *operator checkpoint* as the reliable storage of the current state of the operator instance and the transfer state of all streams produced by this operator instance at a reliable *backup host*. The producer is responsible for keeping the transfer states. Moving this responsibility to the consumer side would cause unnecessary overhead since our DSM model assumes a *multiple consumer – single producer* pattern for data streams. Furthermore, we assume the backup host not to fail. Otherwise, multiple backup hosts are needed to cope with such failures. In a DSMS, the backup host of an operator checkpoint is ideally able to be the host of the corresponding operator. In case of a failure, the backup host which has the operator checkpoint locally available is the destination of operator migration.

4.2. Operator Checkpointing

An operator checkpoint contains the current state of an operator and the transfer states of the outgoing data streams. These states are as introduced in Section 3.2 and categorized by size and frequency of changes in Table 2.

Furthermore, the reliability strategy of the DSMS has to guarantee relaxed consistency (see Section 3.3) in case one or more operators are migrated and restart from their last checkpoint while all other operators keep their current state. In order to reduce the

State	Changes	Size
Operator State (s)	frequent	constant medium
Time Context (TC)	frequent	constant small
Transfer State (TS)	frequent	varying medium – big
Routing Information (RI)	infrequent	constant small

Table 2: Categorization of States

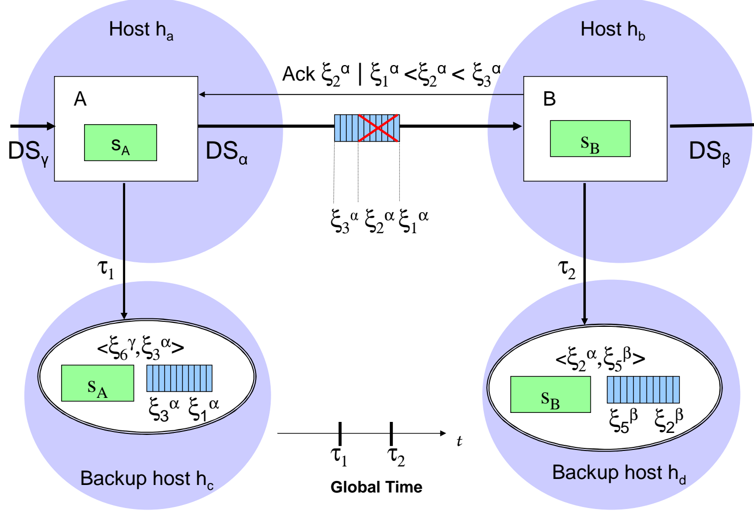


Figure 7: Uncoordinated Checkpointing

effort for checkpointing, we introduce an additional constraint for the reliability strategy according to which only the most recent checkpoint is kept at the backup node. Therefore, it is not possible to go back further than to the most recent checkpoint during operator migration.

4.3. Uncoordinated Operator Checkpointing

Figure 7 illustrates *uncoordinated checkpointing*, where checkpoints are scheduled individually for each operator instance. In the illustrated example, operator A schedules a checkpoint at τ_1. At this point in processing time, operator A has an operator state s_A, a time context $< \xi_6^\gamma, \xi_3^\alpha >$, and a transfer state $TS_\alpha(\tau_1) = DS_\alpha(\xi_3^\alpha, \xi_1^\alpha)$. Later in time at τ_2 operator B schedules a checkpoint with operator state s_B, a time context $< \xi_2^\alpha, \xi_5^\beta >$, and a transfer state $TS_\beta(\tau_2) = DS_\beta(\xi_5^\beta, \xi_2^\beta)$. After performing this checkpoint, operator B will never rollback before this checkpoint so the transfer state of DS_α can be trimmed to the start stream time ξ_2^α which corresponds to the time context of the last checkpoint of B. This is done by sending an appropriate acknowledge message along all input edges after performing a checkpoint.

Uncoordinated checkpointing as described above guarantees relaxed consistency if in case of failures one or more operators are recovered from their recent checkpoints. Based on *relaxed consistency* (see Section 3.3) this guarantees lossless and intra-stream order preserving reliability. A formal proof can be found in [9].

In the *single failure case*, only one operator fails at a time. Without loss of generality, we choose operator B of Figure 7 to fail. In this case, operator B is recovered from the most recent checkpoint taken at τ_2. The connected operators along DS_α and DS_β

are affected by the rollback to the previous checkpoints. Since this was the stream-time of the checkpoint of B, the recovered operator B is able to seamlessly continue to work.

For the *multiple failure case*, we start with a failure of two connected operators. Since consistency is defined pairwise, this argumentation can be extended to general multiple failure cases. Going back to the example, we assume operator A and operator B have been recovered from their recent checkpoints. For the consistency evaluation of this case, we can distinguish between edges to non-failed operators and edges between failed operators. For edges to non-failed operators, we use the argumentation of the single failure case in order to prove correctness of the algorithm. In this case, we have only to investigate for relaxed consistency along the edge (data stream DS_α) connecting the failed operators A and B. For this analysis, we distinguish three cases. Firstly, the checkpoint of operator B was performed after the checkpoint of operator A. In this case, the recovered operator A starts processing earlier in time than the recovered operator B is expecting. Relaxed consistency allows for correct processing in this case. Secondly, checkpoints of operator A and B are synchronous with respect to the stream time of their connecting data streams. In this case, even exact consistency is guaranteed. Thirdly, the checkpoint of operator B was performed earlier in time than the checkpoint of operator A in this case the recovered operator A has to be able to re-send data stream elements for the recovered operator B which are actually before its own recovery time. This is achieved due to recovery of the transfer state and in this case allows to guarantee relaxed consistency.

The presented uncoordinated checkpointing algorithm is suffering from a high transport overhead on sending checkpoint messages from the active host to the backup host. Transfer overhead is defined as the relation between the transport load because of checkpoint messages compared to the transport load of a DSMS without checkpointing. Since the transfer state is part of the checkpoint message the algorithm leads to a high transport overhead. This is particularly expensive for mobile devices because of high energy consumption. In order to supersede this problem, we present in the following another existing reliability algorithm, called ECOC, which aims at coordinating the checkpointing activities of operators.

4.4. Coordinated Operator Checkpointing

The *efficient coordinated checkpointing* (ECOC) algorithm [12] reduces the overhead needed for checkpointing messages between the active and the backup host. A significant portion of the checkpoint message is contributed by transfer states, which are needed to guarantee relaxed consistency. In order to develop an algorithm which allows to omit transfer states from checkpointing, we first analyze in which cases the recovery of transfer states does not have to be considered. The recovery of transfer state is not needed to achieve relaxed consistency in case of failures if it is guaranteed that $\xi_x^{out} \leq \xi_y^{in}$ for every pair of checkpoints along an edge connecting two operators in a stream process.

In order to achieve a temporal coordination of checkpoints, ECOC introduces an additional *checkpoint-request message (Check-Request)* which is sent to downstream operators attached on the corresponding data stream element extending the alphabet of the payload information. Figure 8 illustrates the messages used for checkpoint coordination and the additional *pending checkpoint log* (PCL). An additional advantage of

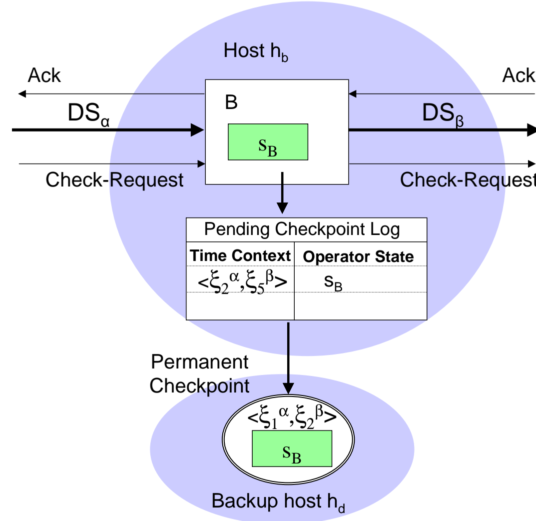


Figure 8: ECOC Overview

this approach is that message exchange is only between connected operator instances in a Peer-to-Peer fashion, without centralized control. The PCL is used to store checkpoints locally until along all outgoing edges all downstream operator have performed their checkpoints and $\xi_x^{out} \leq \xi_y^{in}$ is fulfilled. Further details on the ECOC approach are presented in [12].

For this reason, a two-phase protocol described in pseudocode in Figure 9 is applied. Checkpoints may be triggered by a *local scheduler* or by a Check-Request message from an upstream operator. The local scheduler can follow different strategies for checkpoint planning, e.g., every 50 incoming data stream elements. During execution of the two-phase protocol, the processing of data stream elements by operator instances continues without disturbance.

In the first phase (*planning phase*), a checkpoint is triggered either by receiving a Check-Request or by the local scheduler and stored in the local PCL. Additionally, along all output edges Check-Request messages are sent with the corresponding stream-time context ξ_i^{out}. In the second phase (*checkpoint phase*), corresponding Ack messages with ξ_i^{ack} are received along the outgoing edges. If for a checkpoint in PCL all output edges have received the Ack messages where $\xi_i^{out} \leq \xi_i^{ack}$ the checkpoint is *fully acknowledged*. In this case, the checkpoint is sent to the backup host and removed from the PCL. The PCL is a data structure in the local memory assigned to an operator instance holding a list of checkpoints ordered by time of creation. Following the edges transitively, we see that checkpoint requests are cascaded until they reach an operator instance without internal output data streams. Outside output data streams are ignored in this case. At this operator instance, the checkpoint can be passed immediately to the backup host. The Trim-Ack's are cascaded backwards transitively against the flow of

```
 1   //This code is executed for each operator instance
 2   //this is referencing to this operator instance
 3   while (true)
 4    if (pd = Check-Request) or (local scheduler event) then
 5     //planning phase
 6     if |{e|x = this ∧ y ≠ ow}| > 0 then
 7      add new pending checkpoint do PCL;
 8      send Check-Request's to all output streams with ξ_i^out|i ∈ (0,m] ∧ y ≠ ow;
 9     else
10      do permanent checkpoint;
11      send Ack's upstream ξ_i^in|i ∈ (0,n];
12     endif
13    endif
14    if (Ack ξ_i^ack received) then
15     //checkpoint phase
16     trim transfer state ξ_s = ξ_i^ack;
17     foreach (checkpoint in PCL) then
18      acknowledge checkpoint with ξ_i^ack;
19      if checkpoint is fully acknowledged then
20       save checkpoint permanently at backup host;
21       remove checkpoint from PCL;
22       send Ack with xi_i^in of checkpoint along input edges;
23      endif
24     endforeach
25    endif
26   endwhile
```

Figure 9: Pseudocode of ECOC

data streams and allow to make pending checkpoints permanent at the backup host.

A drawback of the ECOC approach is the delay of a checkpoint in the planning phase. Checkpoints are delayed until all downstream operators have acknowledged the checkpoint. After acknowledgement, the checkpoint message is sent to the backup node. In particular, these delays are getting longer if we go upstream, closer to the sensors, in a stream process. Downstream, closer to the final consumer operators, which themselves have no more output streams, the delays are getting shorter. These delays are not blocking stream processing and have no effect on time constraints in stream time. It has to be noted that checkpoints are always performed when planned. The delay only affects the propagation of the checkpoint to the backup host, but does not impact the regular processing of the data stream. New stream elements produced in the time in which the coordination of a checkpoint takes place will be part of the operator's transfer state. However, when the checkpoint will be actually executed, only those elements of the transfer state that have been produced before the acknowledgement of the checkpoint request will be part of the checkpoint. New stream elements that have been created after the acknowledgement / request of a checkpoint will have to be considered in subsequent checkpoints.

Assuming the case of a failure in the planning phase, the affected operator is recovered from the most recent permanent checkpoint. In this case, correct data stream processing is still guaranteed, but duplicates are produced because of recovering from the older checkpoint. On the other hand, storing checkpoints in the pending checkpoint

log requires additional memory overhead. Since we do not need to store the output queue in the pending checkpoint log, this overhead is similar to the reduced communication overhead. Therefore, we consider these drawbacks as acceptable.

4.4.1. Reliability Levels guaranteed by ECOC

When applying the ECOC formalism for reliability of DSM, we state that ECOC guarantees relaxed consistency for non-cyclic data stream process graphs if in case of failures one or more operators are recovered from their most recent checkpoints. ECOC ensures coordination of checkpoints along outgoing edges which allows the omission of the transfer state in checkpoints ($\xi_x^{out} \leq \xi_y^{in}$). Furthermore, ECOC behaves like uncoordinated checkpointing – which means that it can be proven correct with regard to relaxed consistency (a formal proof can be found in [9]). Furthermore, it is needed to show that ECOC terminates when cascading checkpoint requests transitively along connected edges. This is guaranteed because finally each path in a non-cyclic stream process graph will reach an outside world vertex. The last operator of the DSMS is allowed to perform checkpoints at the backup host immediately. Finally, since the Ack-messages caused by permanent checkpoints are cascaded upstream in the same manner, all pending checkpoints will be acknowledged, which proves the termination of ECOC for non-cyclic stream process graphs.

Delay limitation is not an intrinsic behaviour of the algorithm, except from the fact that cascading checkpoints have to terminate. But delay-limited reliability is achieved by the underlying infrastructure as shown in Figure 6 in Section 3.4. Given the assumption that there is a backup host available which is able to recover the operator within a known recovery time, the infrastructure is able to trigger recovery before the maximum allowed delay time is exceeded. Of course, the maximum allowed delay time has to be reasonably long, given processing and network delays of the investigated environment.

Finally, the ECOC approach is able to support lossless and delay-limited reliability of DSM at operator level with affordable effort.

4.4.2. Extensions of the ECOC algorithm

Supporting complex stream processing topologies is crucial for real-world DSM applications. Recently, research in the area of DSM is focusing on adaptive stream processing [20, 16, 3, 30]. In these research projects, stream processing is continuously adapting to changes in the computing environment, e.g., system load or sensor input characteristics. In general, this implies that a feedback loop is applied within a stream process graph where results of current stream processing are affecting the stream processing processing in the future. In order to model and support such feedback cycles for DSM processing, also reliability algorithms have to support these topologies.

Firstly, we focus on optimizations for join-operators. Particularly for join-operators, obeying all Check-Request messages that may come along different input edges will increase the checkpoint frequency at the operator itself and subsequently on all operators following transitively downstream in the data stream process graph. The checkpoint requests along the different input edges are not correlated and therefore may be received shortly after each other. An increased checkpoint frequency may reduce the benefit achieved by ECOC because resulting again in increased checkpoint overhead.

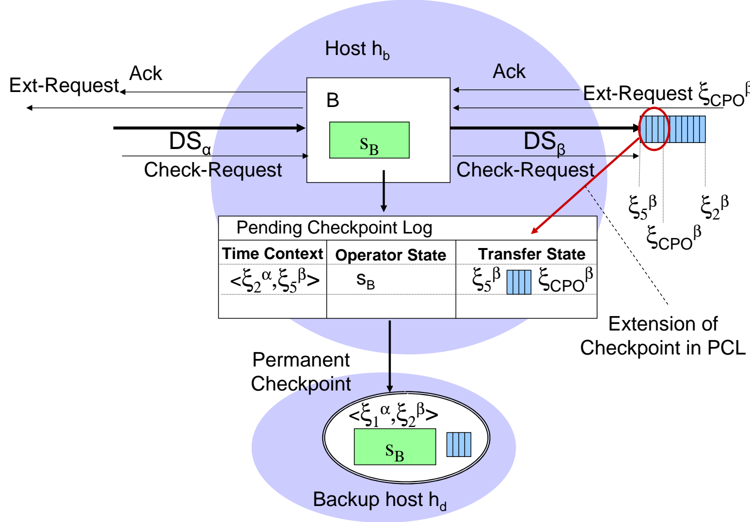


Figure 10: Extended ECOC

In cases where multiple checkpoints are requested in a *short time interval*, it may be beneficial *to extend* the previous still pending checkpoint by the necessary transfer state instead of performing a new checkpoint. Multiple checkpoints in a short time frame may appear due to multiple input edges on operators (as for join operators) or when local scheduling of checkpoints is combined with obeying Check-Request messages from input edges. Due to the coordination of checkpoints in ECOC, a checkpoint imposes load both on the operator itself and on all operators transitively following operators along downstream paths in the data flow of the stream process.

In order to reduce the overhead for checkpoints triggered within a short time frame, we propose an optimized version of ECOC (see Figure 10), where an additional *extension request message (Ext-Request)* is introduced to request the extension of an existing checkpoint in the PCL by a limited part of the transfer state. Adding a subset from ξ_{CPO} to ξ_{CPN} of the transfer state of checkpoint CPN extends the relaxed reliability constraint from $\xi_{CPN} \leq \xi_{ip}^i$ to $\xi_{CPO} \leq \xi_{ip}^i$, where ξ_{CPO} is before ξ_{CPN} which is the time context of the checkpoint to be extended. In Figure 10, ξ_5^β is the stream time of the pending checkpoint with respect to output stream β and ξ_{CPO}^β is the stream timestamp of the Ext-Request. This extension is only applied if the *overall checkpoint load* of the system is reduced compared to the standard ECOC algorithm. Based on this, the extension is only done if the size of the extended transfer state $|TS_e|$ is smaller than the overhead caused by performing a new coordinated checkpoint.

Extended ECOC allows the receiver of a Check-Request message to decide whether a new checkpoint is performed or an Ext-Request is returned to the sender. The decision is based on the average network load imposed by a checkpoint acquired during runtime of an operator instance. From downstream neighbors, each node receives an

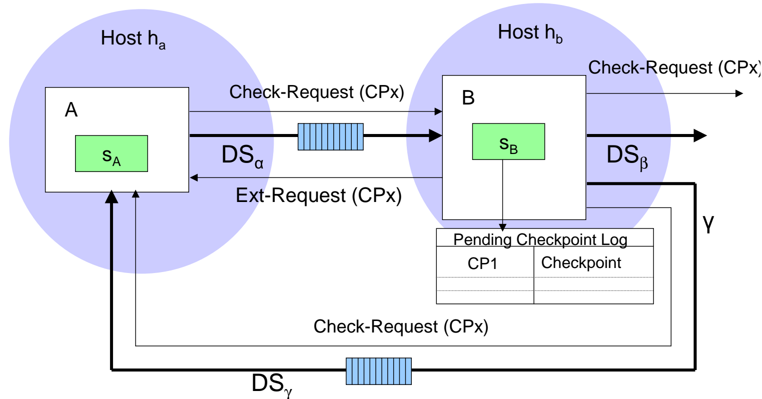


Figure 11: Cycles with Optimized ECOC

average downstream load imposed by all transitive checkpoints triggered along the corresponding output edge. This statistic information is passed upstream as attachment to Ack-messages.

Still, ECOC has to support *cycles* in the stream process. A closed control cycle in a stream process is in particular beneficial in scenarios where stream processing has to adapt dynamically to changes in the data stream characteristics during runtime [20, 16, 3, 30]. For example, the processing of an ECG signal has to be adapted when the heart beat becomes pathologic. In order to adapt stream processing, the operator parameters have to be changed. The cycle support is based on the previous optimized ECOC algorithm. Figure 11 illustrates a cycle in a stream process. The cycle caused an infinite cascading of checkpoint requests with unlimited increase of the pending checkpoint log without applying permanent checkpoints at the backup host. To break this infinite cycle, unique identifiers are applied to Check-Request messages by the first operator instance that triggers the coordinated checkpoint. All transitive checkpoints caused by cascading checkpoints inherit the same identifiers. Consequently, the checkpoint identifiers are also used to identify checkpoints in the pending checkpoint log. Therefore, whenever a Check-Request message is received which has a checkpoint identifier that already is available in the pending checkpoint log, a cycle in checkpoint coordination is detected. After the cycle is detected, the affected node can easily break the request cycle by requesting an extension of the previous checkpoint by using the presented extended ECOC approach. Obviously, the pending checkpoint is extended in this case without regard to checkpoint load statistics.

The proposed optimized ECOC approach is able to support lossless and delay-limited reliability of DSM at operator level with affordable effort [9]. Efficient reliability is achieved for complex stream process graphs including a large number of operators with combinations of splits, joins, and even loops. Extended ECOC is able to adapt its behavior according to acquired data stream statistics in a way that the overall

checkpointing overhead is kept minimal based on Peer-to-Peer communication with neighboring operator instances without establishing a centralized checkpoint control.

5. Evaluation Results

In this section, we present performance evaluations of the coordinated and uncoordinated checkpointing algorithms for mobile devices. The overall objective is to analyze the overhead which is imposed by adding support for reliable DSM, especially in a mobile environment with resource-bound devices, with data stream processes taken from real-world telemonitoring applications. For these evaluations, we use real patient data produced by a single-lead human *electrocardiogram* (ECG) sensor device. We address both the performance during normal (failure-free) runtime of the DSM system and the performance during recovery from one or more failures. The reliable DSM is implemented within our distributed data stream management infrastructure OSIRIS-SE [10, 25, 12, 11]. OSIRIS-SE is fully implemented in Java and therefore only requires a Java virtual machine (JVM) for each participating node. Our evaluation is targeted to measure network transport overhead, CPU load, and memory consumption during the failure-free runtime of a stream process. Due to the requirements demanded by the intended usage for pervasive computing applications including mobile and embedded devices, we consider the utilization of these resources as critical. Moreover, we also investigate the behavior during failure recovery.

This experimental setup consists of four Dell Axim X51v PDAs and three HTC TyTN smartphones. The PDAcs are equipped with an Intel XScale processor with 624 MHz and 64MB of main memory. The smartphones have a Samsung SC32442A processor with 400 MHz and also 64MB of main memory. Both device types are running the Windows Mobile 5 operating system. Each mobile device has a local OSIRIS-SE software layer hosted by an IBM J9 Java virtual machine and all devices are connected via wireless LAN. An additional laptop computer is used to host the OSIRIS-SE global repositories which contain metadata on DSM applications. Due to the peer-to-peer nature of stream process execution with OSIRIS-SE, the laptop computer is not directly involved in stream process execution and does not host any operator instance. The stream processes are executed with a rate of 30 data stream elements per second produced by each sensor operator.

5.1. Evaluation Settings

During the evaluation, stream processes are executed according to four different settings:

1. *Unsafe* stream process execution refers to the execution of a stream process without applying any reliability strategy. In this case, no recovery by operator migration is possible in case of failures. In the unsafe setting, *no operator checkpoints* are scheduled or performed.
2. *Uncoordinated* stream process execution refers to the execution of a stream process based on the uncoordinated operator checkpointing algorithm (see Section 4). This case allows for recovery of failures by operator migration. In this setting, each operator instance triggers checkpointing locally. In our evaluations,

the stream process execution is investigated for different checkpoint intervals c. Checkpoint intervals are specified by means of the number of data stream elements processed between two subsequent checkpoints. For the uncoordinated setting, a fixed checkpoint interval would again cause some form of coordination of checkpoints between the operators. For this reason, checkpointing is not exactly done after every c elements in the uncoordinated setting. At each checkpoint the exact time of the next checkpoint is chosen randomly within an interval from $\frac{c}{2}$ to $\frac{3c}{2}$.

3. *Coordinated* stream process execution refers to the execution of a stream process according to our ECOC operator checkpointing algorithm (see Section 4). This case allows for recovery of failures by operator migration. In the coordinated setting, only sensor operators trigger checkpoints. All other operators receive checkpoint requests via data stream connections. For sensor operators, the checkpoints are scheduled according to the given checkpoint interval parameter.
4. *Extended* stream process execution refers to the execution of a stream process using the extended ECOC operator checkpointing algorithm (see Section 4.4.2). This case allows for recovery of failures by operator migration even for stream process topologies that contain cycles in the flow of data stream processing. Similar to the coordinated setting, also in the extended setting only sensor operators trigger checkpoints and the checkpoints are scheduled according to the given checkpoint interval parameter.

5.2. Investigated Parameters

The following resource utilizations are measured during the failure-free runtime of different stream processes:

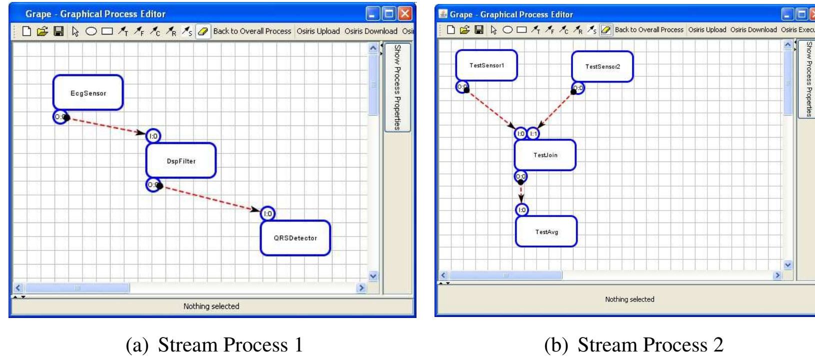
- *Network transport overhead* is the additional amount of communication data caused by uncoordinated and coordinated reliability strategies, respectively. The overhead is measured in relative terms, as ratio of bytes needed for sending checkpoint messages and bytes needed for sending data stream elements between running operator instances.

- *CPU load* is the utilization of the CPU of a participating node during the execution of a stream process.

- *Memory consumption* is the additional amount of main memory of a participating node imposed by uncoordinated and coordinated operator checkpointing during the execution of a stream process. Memory consumption is provided by the JVM. For this reason, some deviations will occur in this measurement due to the heuristics of the JVM's garbage collector.

Furthermore, we evaluate the performance of failure handling of our presented reliability strategies during the operator migration phase:

- *Recovery time* is the time τ_r needed for instantiation of a new operator instance and reconstruction of the recent operator state from the checkpoint (see Figure 6).



(a) Stream Process 1

(b) Stream Process 2

Figure 12: The Evaluation Stream Processes

- *Catch-up time* is the time τ_c needed to work off the congestion that has piled up during the time when the failed operator instance was not available (see Figure 6).
- Resource utilization at the recovering node. The CPU and memory utilization is presented as the average value over the recovery phase τ_r and the catch-up phase τ_c of an operator instance.

5.3. Stream Processes used for the Evaluation

For the experiments, the sample stream process (*SP1*) depicted in Figure 12(a) has been implemented to process real world ECG data within a healthcare application. The *ECGSensor* operator is simulating a sensor for a single-lead human *electrocardiogram* (ECG) by reading real-world data values from a file which have previously been generated by an ECG sensor (in order to make the evaluation repeatable). Each sample within the file contains one float value for the real-world timestamp of measurement and one float value for the ECG voltage. The *DSPFilter* operator is processing the incoming raw-ECG data in order to remove noise. Finally, the preprocessed ECG data stream is arriving at the *QRSDetector* operator. The *QRS complex* is the characteristic shape within the ECG which allow for diagnosis of various diseases of the heart.

The sample stream process *SP2* depicted in Figure 12(b) has been implemented to allow for the analysis of more complex stream processes including a join of two different data streams. This stream process uses artificially generated stream data and corresponds, for instance, to a telemonitoring application which combines sensor streams from a blood pressure and an ECG sensor. Again, the two sensor operators (*TestSensor1* and *TestSensor2*) are generating sensor data streams containing two float values (timestamp and value) for each data stream element (sample). The *TestJoin* operator integrates the two streams by calculating the sum of the data elements from both data streams within a sliding time window with the size of 100 elements. Finally, the *TestAvg* operator is performing an average over a sliding time window with the size of 100 elements on its incoming data stream. Due to the sliding window operations performed

Node	Operators (SP 1)	Operators (SP 2)
PDA 1	ECGSensor	TestSensor1
PDA 2	ECGSensor	TestSensor2
PDA 3	DSPFilter	TestJoin
PDA 4	DSPFilter	TestJoin
Smartphone 1	QRSDetection	TestAvg
Smartphone 2	QRSDetection	TestAvg
Smartphone 3	QRSDetection	TestAvg

Table 3: Operator Providers in Mobile Environment

by TestJoin and TestAvg, the internal state of the operators is larger than in the first stream process.

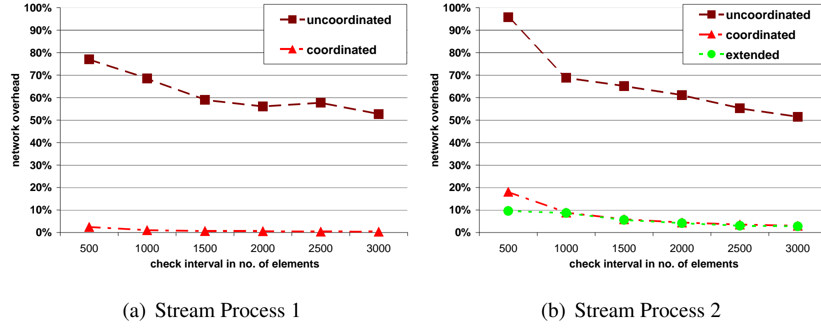
Table 3 illustrates the operators that are available at the different nodes in the experimental setup. Since all operators are available at more than one node in the OSIRIS-SE network, the infrastructure is able to select one node as operator provider and another node as backup provider for each operator instance. The only exception is for SP2 where the backup provider for both TestSensor1 and TestSensor2 is the additional laptop computer. However, this does not affect the measurement because only the providers hosting running operator instances have been evaluated, but not the providers selected for hosting backups. There is also no effect on the failure measurement because no failures are triggered on either TestSensor1 or TestSensor2.

5.4. Evaluation Procedure

The performance evaluation investigates two cases situations of a stream process execution.

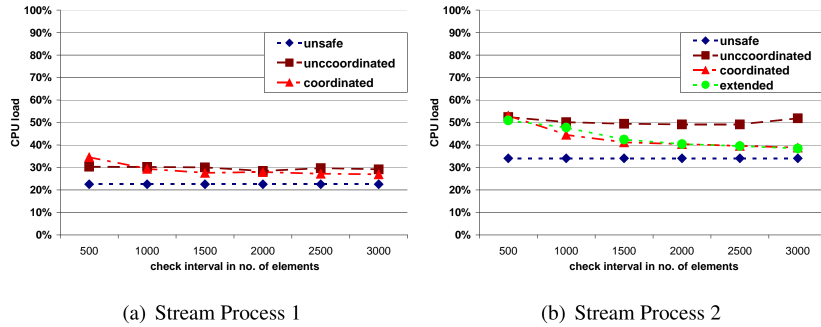
Firstly, the *failure-free runtime* is evaluated. In this case, the stream process is up and running without any failure (such as crashed operator providers or network disconnections). In order to avoid any disturbances, the stream process is executed in a way that each operator instance is hosted by a different provider node. Moreover, each operator instance has a dedicated backup node which is not performing other tasks. The actual selection of provider nodes and backup nodes is done by the OSIRIS-SE infrastructure based on the load distribution at startup time of the data stream process. During the experiment, the stream process is executed for each setting (unsafe, uncoordinated, coordinated, and for SP2 also extended) in combination with the different backup intervals (500, 1000, 1500, 2000, 2500, 3000) for a duration of 400 seconds and averaged logging statistics are collected. Of course, for the unsafe setting the checkpoint interval is not applicable. In order to avoid the influence random disturbances (e.g., operating system tasks) the execution of each measurement is repeated 5 times. The presented results are aggregated over the execution time and the number of repetitions. Moreover, only one stream process is executed at the same time.

Secondly, the *failure handling* is evaluated. In this experiment, the stream process is initially running without failures for a duration of 150 seconds. After that, an operator failure is explicitly triggered. In order to avoid time synchronization effects, a random delay of between 0 and 50 seconds is introduced before the actual failure triggering. As for the failure-free measurement, this measurement is repeated 5 times. Again, only



(a) Stream Process 1 (b) Stream Process 2

Figure 13: Network overhead



(a) Stream Process 1 (b) Stream Process 2

Figure 14: CPU load

one stream process is executed at a time. In order to compare the different strategies, the measurements are performed for each setting (uncoordinated, coordinated, and extended) in combination with the different checkpoint intervals (500, 1000, 1500, 2000, 2500, 3000). Of course, the checkpoint interval is irrelevant in the unsafe setting.

5.5. Evaluation Results

Figure 13 illustrates the evaluation results for the network overhead during a failure-free run of the two sample stream process SP1 and SP2. Comparing the coordinated and uncoordinated setting, we see that ECOC is significantly reducing the overhead for checkpointing due to the transport of checkpoint messages from operator provider to backup provider. Comparing SP1 and SP2, we see a slightly higher network overhead for the coordinated setting. This is because of larger operator states in SP2. For SP2, also the extended ECOC approach has been evaluated because of the join in the stream process topology. For short checkpoint intervals, extended ECOC results in even more reduced network overhead compared to ECOC.

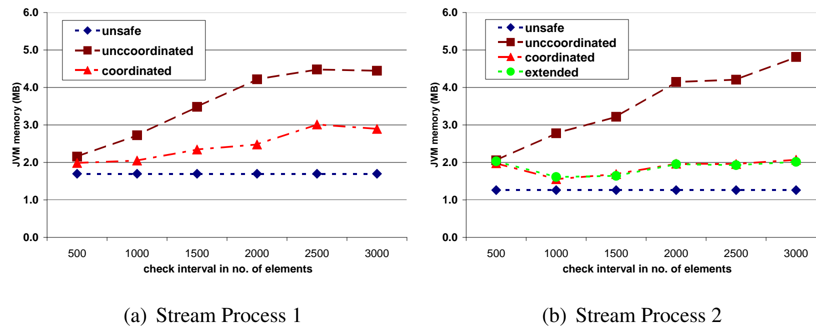
Figure 14 shows the CPU load of the different reliability strategies. Compared to the unsafe setting where no special support for increasing the degree of reliability is applied to stream processing, the overhead of CPU load imposed by coordinated

and uncoordinated operator checkpointing is reasonable. For SP1 which has a simple topology, this overhead is less than for the more complex SP2. Another result indicated by Figure 14 is that the coordinated setting is slightly less CPU demanding than the uncoordinated setting because no transfer state is serialized and sent to the backup host. The extended setting evaluated with SP2 shows no additional CPU overhead compared to the coordinated approach.

Figure 15 illustrates the average JVM memory consumption of a node during stream process execution. Compared to the unsafe setting, the memory overhead imposed by coordinated and uncoordinated checkpointing is reasonable. In particular, the coordinated setting shows only slightly higher memory demand. The significant higher memory demand for the uncoordinated settings comes from the need to also checkpoint the transfer state. During checkpointing of the transfer state, larger data structures in memory are needed to send the larger checkpoint messages. This fact is also pointed out by increasing memory overhead for longer checkpoint intervals which lead to larger checkpoint messages. The extended setting evaluated with SP2 shows no additional JVM memory consumption compared to the coordinated approach.

Failure handling in the mobile environment has been evaluated by triggering an operator failure of the QRSDetector operator for SP1 and of the TestAvg operator for SP2. Figure 16 illustrates the recovery time τ_r of a failed operator instance. Regarding τ_r, there are no significant differences between the different strategies. In addition, there is a slightly shorter recovery time for SP2 which is connected to the higher CPU utilization during recovery in SP2 (see Figure 17), whereas JVM memory consumption (see Figure 18) is almost constant in all settings.

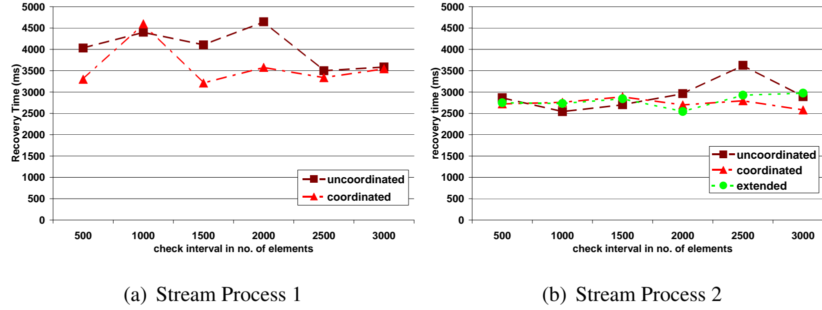
The general belief that frequent checkpoints accelerate recovery is not effective in the mobile environment we considered in our evaluations. There are two reasons for this fact. Firstly, we distinguish between recovery and catchup-time. We define recovery as the process of instantiation of a new operator instance from an existing checkpoint. This time does not depend on checkpoint frequency but on checkpoint size. Moreover, for ECOC even checkpoint size is constant and does not depend on the checkpoint frequency. The catch-up time is defined as the time needed to work off congestion that has piled up after a failure situation has occurred. Therefore, in contrast to



(a) Stream Process 1

(b) Stream Process 2

Figure 15: JVM memory consumption (MB)



(a) Stream Process 1

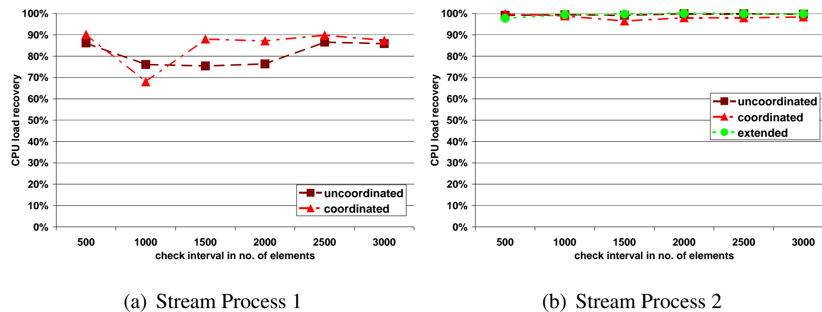
(b) Stream Process 2

Figure 16: Recovery Time (ms)

recovery time, catch-up time is actually related to the checkpoint frequency. Secondly, the CPU load of resource-limited devices in the mobile environment is generally close to full CPU utilization (see Figure 17). The higher CPU utilization at high checkpoint frequencies does not allow for improving recovery or catch-up time.

Figure 19 illustrates the catch-up time τ_c of a failed operator instance. During the catch-up phase, a newly recovered operator instance is working off the congestion that was caused during the time of failure. There are no significant differences between the different reliability stretegies. Also CPU load (see Figure 20) and JVM memory consumption (see Figure 21) are within reasonable variations. Only for higher checkpoint intervals an additional JVM memory overhead is caused for SP1 and the uncoordinated setting. This behavior is similar to the one of the failure-free evaluation.

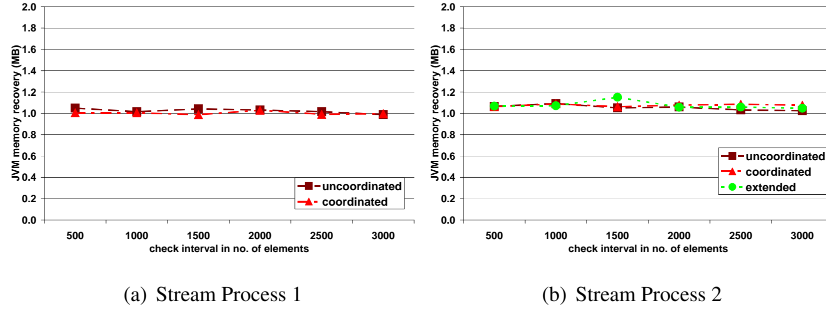
All these evaluations in a mobile environment have shown that the ECOC approach performs significantly better than the uncoordinated setting, which uses the standard passive standby approach. In particular, ECOC dramatically reduces the network overhead, which is the major drawback of the uncoordinated passive standby approach. For short checkpoint intervals, the extended ECOC approach is able to even further reduce the network overhead for certain stream process topologies. Additional measurements



(a) Stream Process 1

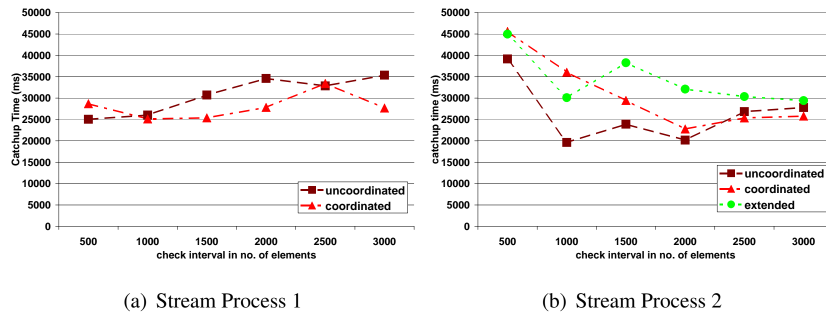
(b) Stream Process 2

Figure 17: Recovery CPU load



(a) Stream Process 1 (b) Stream Process 2

Figure 18: Recovery JVM memory consumption (MB)



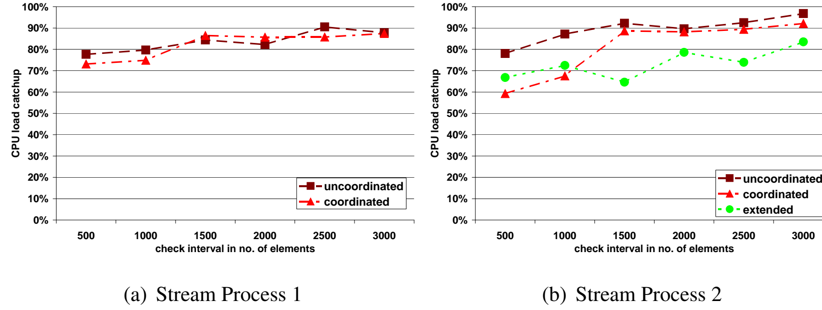
(a) Stream Process 1 (b) Stream Process 2

Figure 19: Catch-up Time (s)

demonstrate that ECOC and extended ECOC do not result in higher memory consumption than the uncoordinated checkpointing approach. For the handling of single and multiple failures, the experiments have shown that ECOC and extended ECOC are about the same in performance compared to the uncoordinated approach with respect to the recovery and the catch-up phase.

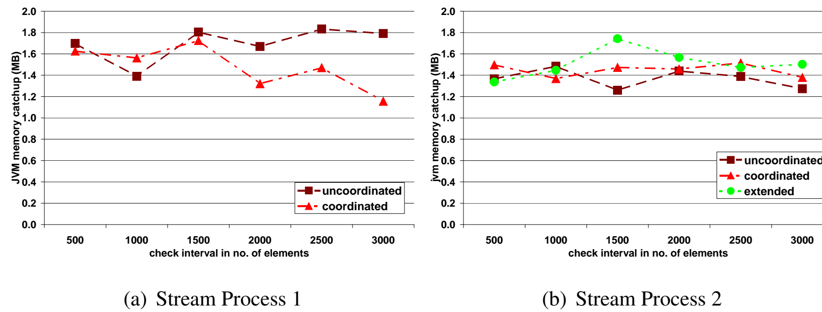
5.6. Further Evaluations in Server Environment

The experimental server environment consists of a network of twelve server nodes. Each node has an Intel Xeon CPU with 3.2GHz, 2GB of main memory, and is running on the Windows Server 2003 operating system. One dedicated node hosts the global OSIRIS-SE repositories. The others are operator and backup providers. All nodes are equipped with a local OSIRIS-SE software layer which is hosted by a Sun J2SE1.6 JVM and thus are able to run operators of the evaluation stream processes. Like in the mobile environment, a node is not operator provider and backup provider at the same time and only the operator providers have been evaluated. All nodes are connected via a reliable Gigabit Ethernet connection. The stream processes within the stationary environments are executed with a rate of 200 data stream elements per second produced by each sensor operator.



(a) Stream Process 1 (b) Stream Process 2

Figure 20: Catch-up CPU load

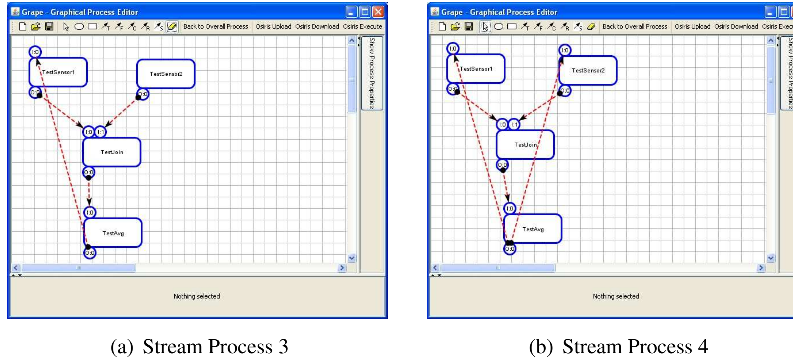


(a) Stream Process 1 (b) Stream Process 2

Figure 21: Catch-up JVM memory consumption (MB)

Figure 22 shows the two additional stream processes, SP3 and SP4, that have been evaluated in the server environment in order to also analyze the effect of the extended ECOC approach for cyclic graphs and for joined data streams. Additional, feedback cycles from the *TestAvg* to one or both *TestSensor* operators allow to influence the sensor processing. In these sample processes, generated sensor values are attenuated if the result of *TestAvg* exceeds a certain threshold. This can be used to achieve detailed processing when a critical health condition appears.

As already shown in the mobile environment, the network overhead (Figure 23) of the uncoordinated setting is significantly higher compared to the coordinated (ECOC) approaches. Regarding the influence of the backup interval, we see no significant increase of CPU load for smaller backup intervals. A reason for this is the fact that the CPU load in the server environment was generally low, especially compared to the evaluations done on mobile devices. Another reason is that performing a backup is not causing significant additional load to the CPU compared to processing 500 incoming data stream elements as for the shortest backup interval in our evaluation. This work focuses on reduction of network overhead and not reduction of CPU load because network overhead is more resource demanding (i.e., energy demanding) than CPU overhead. When further analyzing the uncoordinated setting for the various stream pro-



(a) Stream Process 3

(b) Stream Process 4

Figure 22: Additional Loop Processes

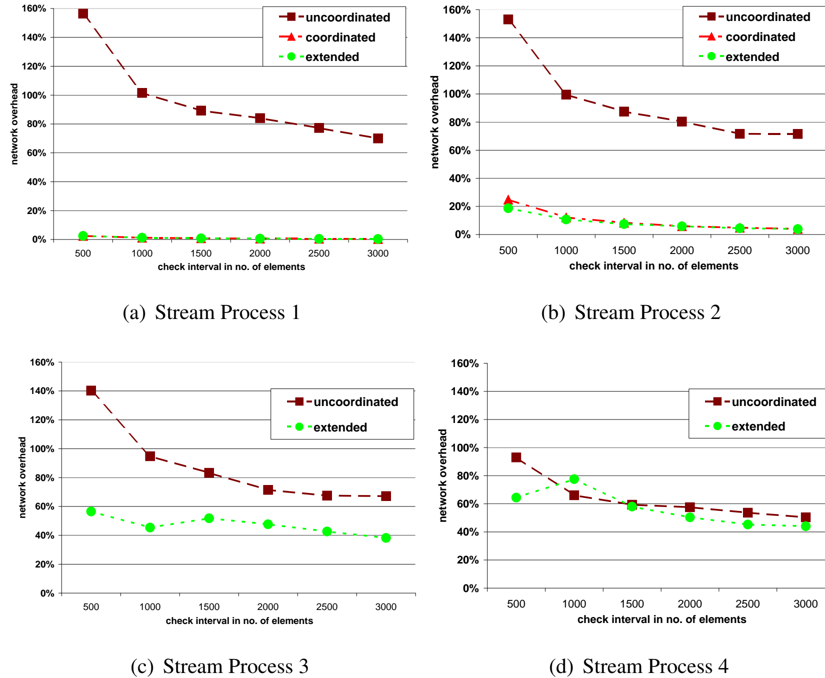
cesses, we see that for SP3 and SP4 the network overhead is reduced compared to SP2. The reason for this is that the network overhead is measured as ratio of bytes needed for sending checkpoint messages and the number bytes needed for sending of data stream elements. In SP3 and SP4 the number of bytes needed for checkpoint messages has only moderately increased compared to the number of additional bytes needed for sending data stream elements along the new feedback data streams. This results in less network overhead for a more complex stream process in the uncoordinated setting.

Figure 24 illustrates the CPU load of the different reliability strategies in the server-based evaluation. The experiments show that there is no measurable CPU overhead due to uncoordinated and coordinated operator checkpointing, compared to the unsafe setting. In general, CPU utilization increases with more complex stream process topologies. In some experiments, the unsafe setting even is slightly more CPU demanding compared the three reliable settings – which might be based on the higher network utilization in the reliable settings that, in turn, result in idle CPU cycles due to network access.

Moreover, the experiments have shown that OSIRIS-SE implements reliable DSM processing for various platforms in a resource efficient way. Overheads due to reliability algorithms for data stream processing are affordable even for mobile devices. By means of the the ECOC algorithm, network overhead can be minimized. Furthermore, the experiments have shown that memory overhead is the second major overhead due to reliability algorithms. Nevertheless, increasing memory availability even for mobile devices is relieving this issue. CPU overhead for reliability is still measurable but can be tolerated as it imposes only moderate additional load.

6. Related Work

The presented reliability approach is based on process-pairs [7], that describes a model of primary and backup processes. The primary process checkpoints all requests to the backup process, so that the backup process has all information necessary to take over control in case the primary process fails. This approach has been widely adopted



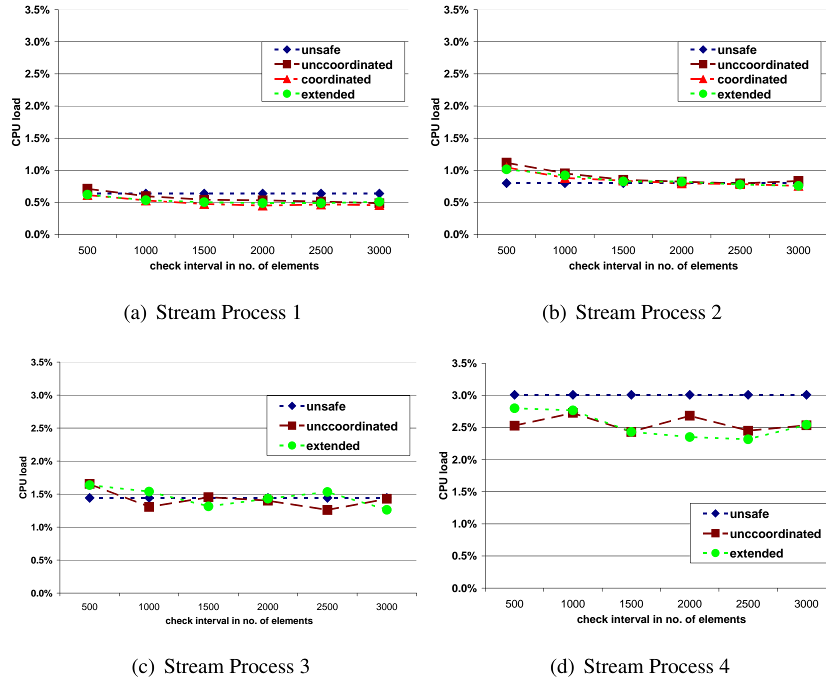
(a) Stream Process 1 (b) Stream Process 2

(c) Stream Process 3 (d) Stream Process 4

Figure 23: Network overhead in server environment

in distributed database research, for example in the Tandem architecture [8]. Furthermore, work of Chandy and Lamport [15] emphasized on performing checkpoints in a distributed system in order to get a meaningful global state. Based on this work, Baldoni et al. [6] have designed algorithms to take consistent checkpoints for general distributed systems. Elonazhy et al. [17] have presented a survey of work on rollback recovery protocols in message-passing systems. As distributed DSM systems are special kinds of message-passing systems, many of these protocols can be applied to DSM as well. However, effects like the domino effect cannot occur in our approach. Our DSM model only keeps track of the most recent checkpoint in order to reduce the overhead. Therefore, rollback propagation –a prerequisite for the domino effect– is not possible. Still, the analyzed reliability algorithms guarantee that all most recent checkpoints are consistent and support lossless reliability. This is achieved by relaxing the notion of reliability with respect to our deterministic system model.

Only few approaches address aspects of reliability in DSM – although this area has become increasingly popular in the last few years. Unlike most of the work in this field, we focus on reliability of DSM at the level of data stream operators. Temporary network disruptions can be addressed by buffering stream elements between pairs of subsequent operators (e.g., STREAM [21], PIPES [13], or GSN [1]). Algorithms for reliable DSM have been discussed in [19] as part of the Aurora [4] project. In contrast to our work, Aurora focuses on the reliability of a whole DSM engine running at



(a) Stream Process 1

(b) Stream Process 2

(c) Stream Process 3

(d) Stream Process 4

Figure 24: CPU load server environment

a node within a loosely-coupled network, whereas we focus on reliability at operator level. Similarly, the process-pairs approach and checkpointing is applied, but without the focus on taking coordinated consistent checkpoints at operator level. Further work [5] presented in the context of Borealis [4], an extension of Aurora, allows for reduced result quality which is not applicable in applications where lossless reliability levels are demanded (e.g., in healthcare). Work in the context of the TelegraphCQ [14] project is providing fault tolerance and load balancing by applying parallel data stream processing [28]. In this approach, multiple instances of the same operator are running in parallel at different hosts. Flux partitions the overall data stream in an adaptive way across this multiple instances. Our work focuses on mobile environments where resources are limited and therefore such active process-pairs approaches are not applicable. Other work [31] in the area of DSM is also investigating the migration of stateful DSM operators but with the focus on optimization of DSM rather than on achieving reliability.

Work in the area of reliability in sensor networks mainly focuses on data transport reliability [29, 23]. Outcome of this research is also beneficial for reliable DSM, but not sufficient. Reliability is also needed at the higher level of abstraction for data stream operators.

Approaches in the area of reliable middleware [27, 22] propose central coordinators in a process-pairs approach and discrete, non-streaming function calls. The focus of

this work is on server clusters, rather than on mobile environments.

7. Conclusion

Applications which operate on data that is continuously generated by hardware and/or software sensors require proper system support for data stream management (DSM). In most cases, like for instance in health monitoring, reliable DSM is a crucial requirement. In this paper, we have introduced a model for DSM where data stream operators are distributed across a loosely coupled network of hosts. Based on the DSM model, we have identified failures that are likely to occur in a DSM system during runtime. Furthermore, as one major contributions of the paper, we have formally defined reliability levels of DSM, based on input/output behavior of DSM systems which are considered black boxes as seen from the outside world. The reliability levels, in turn, allow for a precise characterization of the consistency a DSM system is able to provide at runtime. The three dimensions along which the reliability levels are defined address i.) limited-loss and lossless DSM, ii.) limited-delay and delay-free DSM, and iii.) intra-stream and inter-stream order preservation.

Moreover, we have presented an algorithm for efficient coordinated checkpointing, ECOC, based on this DSM model. ECOC provides consistent operator checkpoints with low overhead. It is based on the migration of stateful data stream operators. In order to allow operator migration in case of failures, a recent checkpoint of the state is needed. It has been shown that ECOC meets the lossless reliability level which is particularly important for medical applications in which each single data element of a data stream might be highly relevant to characterize the physical situation of a patient.

Finally, we have provided a thorough evaluation of the existing ECOC approach in a mobile environment. A prototype implementation of a distributed DSM system, called OSIRIS-SE, has shown a significant reduction of the overhead for checkpointing, compared to standard uncoordinated checkpointing, while still keeping the desired lossless reliability level. This allows for reliable DSM with reasonable overhead even on mobile devices.

Future work on the DSM system OSIRIS-SE aims at further extending the self-adaptation and self-healing capabilities of the system, e.g., by dynamically deploying operator instances on devices with free resources to make best usage of mobile and fixed devices in a network.

[1] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In *Proceedings of the 8th International Conference on Mobile Data Management (MDM)*, pages 198–205, Mannheim, Germany, May 2007.

[2] H. H. Asada, P. Shaltis, A. Reisner, S. Rhee, and R. C. Hutchinson. Mobile Monitoring with Wearable Photoplethysmographic Biosensors. *IEEE EMB Magazine*, 22(3):28–40, 2003.

[3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 407–418, June 2004.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 2004.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD Int'l Conference on Management of Data*, pages 13–24, Baltimore, MD, USA, 2005.

[6] R. Baldoni, F. Quaglia, and P. Fornara. An Index-Based Checkpointing Algorithm for Autonomous Distributed Systems. *IEEE Transactions on Parallel Distributed Systems*, 10(2):181–192, 1999.

[7] J. Bartlett. A NonStop Kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 22–29, Asilomar, CA, USA, 1981.

[8] J. Bartlett, J. Gray, and B. Horst. Fault Tolerance in Tandem Computer Systems. Technical Report TR 86.2, Tandem, 1986.

[9] G. Brettlecker. *Efficient and Reliable Data Stream Management*. PhD thesis, University of Basel, 2008. `http://dbis.cs.unibas.ch/publications/2008/phd-thesis-gert-brettlecker/`.

[10] G. Brettlecker and H. Schuldt. The OSIRIS-SE (Stream-Enabled) Infrastructure for Reliable Data Stream Management on Mobile Devices. In *Proceedings of the SIGMOD Int'l Conference on Management of Data*, pages 1097–1099, 2007.

[11] G. Brettlecker, H. Schuldt, and R. Schatz. Hyperdatabases for Peer–to–Peer Data Stream Processing. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 358–366, San Diego, CA, USA, 2004.

[12] G. Brettlecker, H. Schuldt, and H.-J. Schek. Efficient and Coordinated Checkpointing for Reliable Distributed Data Stream Management. In *Proceedings of the 10th East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 296–312, Thessaloniki, Greece, 2006.

[13] M. Cammert, C. Heinz, J. Krämer, B. Seeger, S. Vaupel, and U. Wolske. Flexible Multi-Threaded Scheduling for Continuous Queries over Data Streams. In *Proceedings of the First International Workshop on Scalable Stream Processing Systems (SSPS'07)*, pages 624–633, Istanbul, Turkey, Apr. 2007.

[14] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the First Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.

[15] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, 1985.

[16] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Adaptive Stream Filters for Entity-based Queries with non-value Tolerance. In *Proceedings of the Int'l Conference on Very Large Data Bases (VLDB)*, pages 37–48, September 2005.

[17] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comp. Surv.*, 34(3):375–408, 2002.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st Int'l Conference on Data Engineering (ICDE)*, 2005.

[20] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 1338–1341, September 2005.

[21] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *First Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, 2003.

[22] P. Narasimhan, T. A. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for Real-Time Fault-Tolerant CORBA: Research Articles. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[23] J. Paek and R. Govindan. RCRT: Rate-controlled Reliable Transport for Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 305–319, 2007.

[24] S. Petersen and V. Peto and M. Rayner and J. Leal and R. Luengo-Fernandez and A. Gray. The European Cardiovascular Disease Statistics. *European Heart Network*, 2005.

[25] H.-J. Schek and G. Brettlecker. Research Issues in Pervasive Information Management for Health Monitoring and e-Inclusion. In *Proceedings of the First European Conference on eHealth (ECEH)*, pages 11–12, 2006.

[26] Sensatex. The SmartShirt System. `http://www.sensatex.com`, 2007.

[27] S. Seshadri, L. Liu, B. F. Cooper, L. Chiu, K. Gupta, and P. Muench. A Fault-Tolerant Middleware Architecture for High-Availability Storage Services. In *IEEE Int'l Conference on Services Computing (SCC)*, pages 286–293, 2007.

[28] M. Shah, J. Hellerstein, and E. Brewer. High Available, Fault-Tolerant, Parallel Dataflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 827–838, 2004.

[29] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proceedings of the International the Workshop on Sensor Network Protocols and Applications*, pages 102–112, Anchorage, Alaska, USA, 2003.

[30] Y. Tu, M. Hefeeda, Y. Xia, and S. Prabhakar. Control-based Quality Adaptation in Data Stream Management Systems. In *Proceedings of the Int'l Conference on Database and Expert Systems Applications*, pages 746–755, August 2005.

[31] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proceedings of the SIGMOD Int'l Conference on Management of Data*, pages 431–442, Paris, France, 2004.