Proceedings of Datenbanksysteme in Business, Technologie und Web (BTW) 2005,
11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"
(c) Bonner Köllen Verlag: GI-Edition - Lecture Notes in Informatics (LNI), P-65, 2005

Towards Reliable Data Stream Processing with OSIRIS-SE

Gert Brettlecker, Heiko Schuldt, Hans-Jörg Schek

University for Health Sciences, Medical Informatics and Technology (UMIT)
Hall in Tirol, Austria
firstname.lastname@umit.at

Abstract: Recent advances in sensor technologies, wireless communication standards, powerful mobile devices and wearable computers strongly support novel types of applications. Especially in healthcare, telemonitoring applications will make use of this new technology in order to improve the quality of treatment and care for patients and the elderly. These telemonitoring applications require an infrastructure that is able to efficiently combine, process, and manage continuous streams of data coming from the different sensors. In healthcare where applications can be life saving, a particularly important requirement is that this infrastructure is highly reliable. In this paper, we introduce OSIRIS-SE, a novel information management infrastructure for data stream processing based on the hyperdatabase OSIRIS that features a high degree of reliability.

1 Introduction

Recent trends in ubiquitous and pervasive computing, together with new sensor technologies, wireless communication standards, powerful mobile devices and wearable computers strongly support novel types of applications. Especially in healthcare, telemonitoring applications will make use of this new technology in order to improve the quality of treatment and care for patients and the elderly. Chronic ailments such as cardiovascular diseases, hypertension, and diabetes affect a significant number of the western population [AHA01]. In particular, if we consider our aging society, the amount of elderly people suffering from one or more chronic diseases is increasing. Telemonitoring applications enable healthcare institutions to take care of their patients while they are out of hospital, which is especially useful for managing various chronic diseases as well as for measuring the effects of treatments under real-life conditions. A similar but even more comprehensive application is the support for elderly people living at home. Here, not only physiological sensors and data are relevant for telemonitoring but also context information, e.g., information on what a person is currently doing, where she or he is located, etc. The term *eInclusion* addresses the improved embedding of elderly people into the society by means of information technology to improve their quality of life.

Telemonitoring applications supporting data stream management (DSM) require an infrastructure that is able to efficiently combine, process, and manage continuous streams of data coming from the different sensors. Especially in healthcare and eInclusion, a vital require-

ment is that this infrastructure is highly reliable, since it can potentially be life-saving. In this paper, we introduce a novel information management infrastructure that supports DSM with a high degree of reliability. Concepts of Peer-to-Peer computing, grid computing, process support, and replication management are the foundations of our proposed infrastructure OSIRIS-SE. This infrastructure is particularly focused to support telemonitoring applications in healthcare, but not limited to this domain. Similar requirements that arise in other areas, e.g., in environmental monitoring or in traffic control systems, can also be supported.

1.1 A Sample Application of DSM in Healthcare

The following telemonitoring scenario will serve as illustration for the needs of a highly reliable DSM infrastructure and will be used throughout the course of this paper. Fred, aged 71, has already been suffering from *congestive heart failure (CHF)* for 1.5 years. CHF is a disorder causing the heart to lose its ability to pump sufficient blood through the body. Additionally, Fred has chronically high blood pressure which is related to CHF. Due to these facts, Fred has a high risk of heart failure. Fred's treatment includes many emergency department visits and hospitalizations, which have physical and emotional impact. Even worse, since Fred is living in a rural area, the transportation needed for these consultations is quite time consuming and causes additional troubles. In this scenario, disease management can only be improved by reduced time intervals between physician visits. Even then, critical situations might occur between consultations and are usually not detected in time.

Given an appropriate sensor technology and a reliable infrastructure for telemonitoring and data stream processing, i.e., an infrastructure their users can count on, Fred's caregiver will immediately decide to equip him with a wearable telemonitoring system consisting of ECG, blood pressure and oxygen saturation sensors attached to Fred's body, e.g. sensors in a smart shirt [PMJ02]. A PDA carried by Fred then offers computational power and wireless communication. In order to interpret Fred's vital signs, appropriate additional context information is needed, e.g. ECG signals vary if Fred is running or sleeping. Inbuilt accelerometers and information captured by Fred's smart home environment, e.g., an electronic scale, position sensors, or a medication dispenser, provide this context information. The caregiver is able to define individual data stream processing for Fred in a graphical "boxes and arrows" approach by combination of basic building blocks.

1.2 Reliable Telemonitoring Infrastructure

Fred's vital sensors and context information is processed by a telemonitoring infrastructure, that will analyze the data accumulated, and that extracts and forwards relevant information to the care provider in charge. The increased number of components, devices, and platforms in this field of application leads to an increased failure probability. Reliability

is of utmost importance in telemonitoring applications. The infrastructure has to provide a flexible platform for different kinds of monitoring applications and has to guarantee a high level of reliability. In this paper, we present such a reliable telemonitoring infrastructure for data stream management we have developed based on the OSIRIS hyperdatabase infrastructure. Although DSM receives increasing popularity among researchers, only a few projects particularly emphasize on reliability aspects of DSM [SHB04, HBR+03]. In this paper, we introduce new reliability strategies for DSM.

Using this infrastructure, the health status of Fred can be monitored online, as discussed above. In case of relevant changes of Fred's health condition, his physician in charge will be automatically informed, and is able to retrieve all medical important data. Also, the infrastructure will detect critical events, which require immediate intervention (e.g., heart attack, fall, or unconsciousness), and will invoke reliable processes for calling the emergency service. As a consequence, the infrastructure must offer near real-time processing, manage large volumes of continuously produced data, and trigger timely responses of processing results in a reliable way. As outcome of this, Fred's disease will be better managed with less hospitalization and a higher quality of life.

Outline of this Paper: In section 2, we introduce the basics of our infrastructure for DSM, called *OSIRIS-SE*. Section 3 defines reliability of DSM and introduces classes of failures in a distributed DSM system. In section 4, we describe the algorithms OSIRIS-SE implements to achieve a high degree of reliability for data stream management. Section 5 surveys related work in this field and Section 6 concludes.

2 Data Stream Processing with OSIRIS-SE

In this section, we briefly introduce the *hyperdatabase* infrastructure on which our DSM solution has been built and we introduce basic notions and notations.

2.1 Process Management of the OSIRIS-Hyperdatabase

Since in most domains specialized application systems or components are well in place, applications are no longer built from scratch but rather by combining services of existing components into cross-application processes or workflows. These components implement their own data management and provide application services to the outside. Hence, similar to supporting access to shared data, current infrastructures need to provide access to shared services. *Hyperdatabases (HDB)* [SBG+00, SSSW02, SSW02] offer this infrastructure that supports the execution of processes on top of distributed components using existing services. The HDB provides transactional guarantees for processes, sophisticated routing strategies to dynamically choose among the available components, and allows to balance the overall load among them. Hence, the HDB provides database-like functionality, but not at the data level but at the level of access to services.

The *OSIRIS* (Open Service Infrastructure for Reliable and Integrated process Support) system [SWSS03] is a prototype implementation of a hyperdatabase infrastructure that has been developed at ETH Zurich. OSIRIS controls the execution of *traditional processes*. We refer to traditional processes as ordered sets of well defined activities which correspond to the invocation of (web) services in a request/reply style. Due to the discrete invocations, these processes are not able to perform data stream processing. Furthermore, OSIRIS allows for reliable and distributed process execution in a Peer-to-Peer fashion without centralized control. According to this, a node of the OSIRIS network works off its part of a process based on locally replicated meta-data about the process definition and then directly migrates the process instance data to nodes offering a suitable service for one of the next steps in the process. For this reason, process navigation is decoupled from meta-data replication. Hence, the infrastructure is able to scale well with the number of processes. By distributing service requests within the overall system according to the current load status of providers, process execution can be dynamically adapted (e.g., in case several semantically equivalent services exist, the provider with the least approximated load is chosen; this decision is again based on locally replicated load information).

For this reason, the OSIRIS architecture consists of two parts. Firstly, a local software layer is running on each node, called *HDB-Layer*. Secondly, core services offer repositories for process definitions, available services and provider, load status of services, to name only the most important. The HDB-Layers can subscribe for parts this global information, which are locally needed for process execution. By following the concept of transactional processes [SAS99], the execution of processes can be made reliable, even in case of failures and concurrent access to shared services. More on the OSIRIS infrastructure, can be found in [SWSS03, SWSS04].

At UMIT, we have extended the OSIRIS hyperdatabase infrastructure to support data stream processing. This DSM infrastructre is called *OSIRIS Stream Enabled (OSIRIS-SE)*.

2.2 Data Streams

A *data stream* is defined as a continuous transmission of data elements. Each data element contains several data items as payload information. Data elements have a discrete time, which is realized by designated data items containing *sequence numbers* (stream relative). Sequence numbers allow for the recognition of missing data elements and correct ordering in a stream. Additional data items for time-stamps have the advantage to allow for time correlation between data streams.

2.3 Operators

Operators perform the processing steps of data stream management. Running instances of operators are called *operator instances*. In the remainder of the paper, operator is used as short notation for an operator instance. A node of the OSIRIS-SE network hosting a run-

ning operator is also called *provider*. Providers offer the ability of executing an operator as a service to the infrastructure. *Operator classes* implement different semantics of stream processing (e.g., operator class A realizes noise reduction on ECG sensor readings.). One operator class may have multiple *operator types*, which all have the same processing semantics, but different device requirements (processor load, memory consumption, power consumption, network bandwidth, etc.) and also different quality of service parameters (precision, delay, etc.). Running operator instances process incoming data streams from *upstream operators* according to the operator type and feed the results into outgoing data streams for *downstream operators*. Therefore the infrastructure handles queues for each input and output data stream of an operator. Also, the infrastructure applies the sequence numbers to outgoing data elements. The main difference between processing data streams and service invocations of traditional processes is that data streams have a sequence in time. This sequence of elements has to be processed in the appropriate time order. The outgoing data elements again belong to a sequence of elements also called data stream. Due to this nature of data streams, operators keep and aggregate an *operator state* during their life-time of processing. In addition to the task of continuously consuming input and producing output data stream elements, operators access external systems, e.g., a database for storage of relevant processing results. The results on external systems are called the *side effects* of an operator.

2.3.1 Sensor/Output Operators

Sensor operators are operators without input data streams. These operators acquire their input data stream directly from sensors or external systems. Output operators are operators without output data streams. These operators store or transmit the result of stream processing to external systems. This interaction with external systems is a side effect of the operator. Revisiting our initial application scenario with patient Fred, the ”ECG Acquisition” operator of the process depicted in Fig. 2 is a sensor operator, which acquires Fred’s ECG signals. The ”Critical Detection” operator in the same figure is an output operator, which invokes an alarm process in case a critical health status is detected.

2.3.2 Stateful/Stateless Operators

From a processing point of view, *stateful operators* need to aggregate a processing state in order to perform their processing. *Stateless operators* do not need to access state information in order to perform their processing. From an infrastructure point of view, both classes of operators have an attached operator state. The operator state of a stateless operator has no processing state. Details about the operator state follow in the next section. Revisiting our initial scenario, the ”ECG Variability” operator of the process in Fig. 2 is an example of a stateful operator, which calculates the variability of Fred’s ECG signal over a defined time interval. The time interval is defined by the process user, i.e., Fred’s physician.

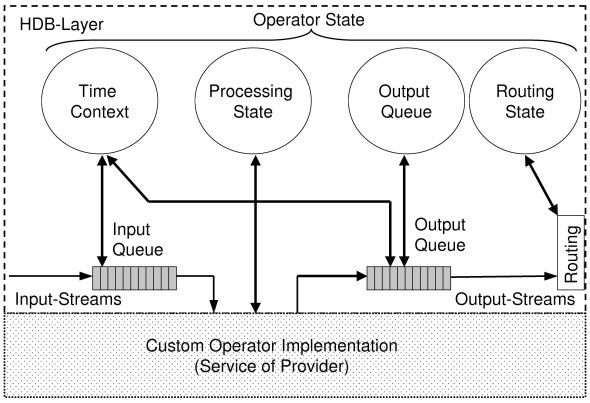

Figure 1: OSIRIS-SE Operator

2.3.3 Operator State

Some information about a running operator instance, called operator state, is kept during the life-time of an operator and is necessary to perform DSM. This operator state handling is offered by the infrastructure layer of OSIRIS-SE (see Fig. 1). The operator developer does not need to care about these details. The state of an operator consists of the following parts:

Time Context: The relative time context of data stream elements is set by attached sequence numbers. The operator instance shall not process the same data stream element twice, therefore the infrastructure has to remember the last sequence number processed for each input stream. Additionally, sequence numbers allow for the infrastructure to detect if each element has been processed in order to avoid gaps in stream processing. Also data stream elements produced for outgoing streams have consecutive sequence numbers whereas the last produced sequence number per each outgoing stream is part of the operator state.

Processing State: Stateful operators require to aggregate a processing state. For example, a windowed average calculation requires to sum all data stream elements within the time window. After processing a bunch of data stream elements, the running operator instance entrusts the processing state to the infrastructure layer for safekeeping. This is done by offering the operator designer an interface to the HDB-Layer. If the next bunch of stream elements is ready for processing, the running operator instance access the aggregated processing state again.

Output Queue: Output queues contain processed data stream elements for downstream operator and are also part of the operator state. Data stream elements are removed from the output queue, if no downstream operator relies on them. This queue trimming mechanism is described in Section 4.3.

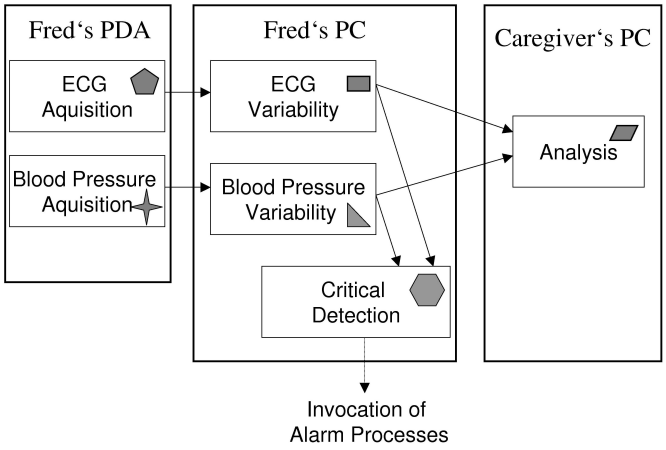

Figure 2: Stream Process for the Sample Application of 1.1

Routing State: The infrastructure has to know the destination of outgoing data streams, which is also stored together with the state of the operator.

2.4 Stream Processes

A *stream process* is a well defined set of logically linked operators continuously processing the selected input data streams, thereby producing results and having side effects. Optional outgoing data streams can be used as input streams for other stream processes. Operators are similar to activities or single service invocations in traditional process management. Figure 2 illustrates a stream process of Fred, which continuously monitors Fred's ECG and blood pressure. Stream processes are designed by *process developers*. In our application scenario, this can be doctors or application developers with a medical informatics background in a medical center. In addition, *process users* which deploy and execute stream processes, are able to customize the stream process before execution according to their needs. This customization includes *quality of service parameters* in order to define the accuracy, time-constraints, accepted reliability and priority of the process. Also parameters of operator processing are customized by the process user. For example, in Fred's stream process, thresholds for the critical detection have to be set individually by his physician.

2.5 Stream Process Execution of OSIRIS-SE

Data stream processing of OSIRIS-SE [BSS04] is the continuous execution of stream processes as described above. Similarly to the OSIRIS process execution, the execution of stream processes is based on locally replicated meta-data. Additional information in core

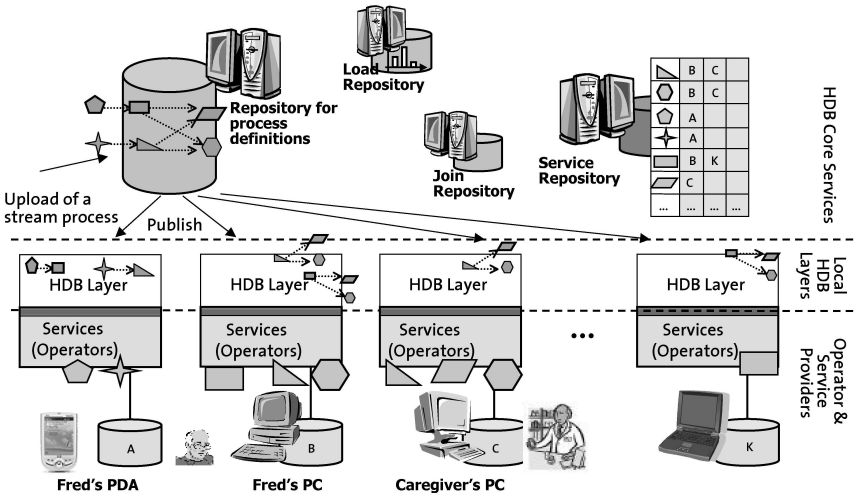

Figure 3: OSIRIS-SE Architecture

repositories is needed for stream processing, like available operators and providers, stream process definitions, and location of join nodes. The architecture of OSIRIS-SE is shown in Figure 3. This figure also illustrates the local replication of a stream process definition.

The life-time of a stream process consists of three phases. Firstly, during the *activation phase*, the necessary operator instances of the stream process are activated. This activation can be seen as a traditional process executing single service invocations, like a request for operator activation. If the operator has output data streams, this activation request must contain the routing state in order to know destinations of output streams. The routing of an operator is done by the HDB-Layer running the upstream operator instance. In case of multiple upstream operators which exist for a join node, one of the upstream operators is marked with a routing flag. The HDB-Layer hosting the marked operator is responsible for routing and its routing decision is propagated to the other upstream operators. Secondly, during the *running phase* of the stream process all necessary operator instances are active, process the input streams, generate output streams, and produce side effects. Finally, during the *deactivation phase*, a shutdown of a stream process is performed, and all running operator instances of the process are deactivated. More on data stream processing with OSIRIS can be found in [BSS04].

Revisiting our initial scenario, the stream process of Fred Fig. 2 is activated by his physician, after applying all necessary body sensors. During the running phase, the stream process monitors Fred's ECG and blood pressure signals in order to detect critical situation and allow for the analysis by the caregiver. After the telemonitoring treatment, the stream process is deactivated.

Obviously, stream process execution requires sophisticated failure handling. We assume that failures might happen in all phases of the stream process's life-time. A detailed discussion of failure handling in OSIRIS-SE follows in Chapter 4. The next chapter defines reliability and availability of stream processing and failure classes.

3 Reliability and Failures of Data Stream Processing

In this section, we introduce our understanding of reliability in data stream processing as well as basic terms in this context. In addition, we discuss relevant failure classes.

3.1 Reliability of Data Stream Processing

In our understanding of reliable data stream processing, stream processes have to be executed in a way that the process specification is met, even in case of failures, i.e., to correctly execute all operators in proper order. Failing to fulfill this specification and possibly optional quality of service requirements set by the user indicates *incorrect data stream processing*. Our main focus is driven by reduction of these failure events, which means to increase the reliability of the system. Reliability is defined based on the probability in which failures occur. Availability is a different important aspect, where the downtime of the system is considered. Availability is defined as the ratio of downtime to overall runtime. Revisiting our medical telemonitoring scenario, we find that short downtimes, that can be shielded by the queues, are tolerable. If longer downtimes occur, Fred and his caregiver have to be informed about the situation so that they can act appropriately. This scenario does not allow low reliability, where incorrect processing results are produced without knowledge of Fred or his caregiver. The loss of critical processing results may have serve consequences on Fred's health status, e.g. critical health conditions are not detected by his stream process of Fig. 2.

Therefore, reliability of data stream processing is tightly coupled to quality of service parameters defined by the process user and process designer respectively. Our infrastructure is designed to meet these requirements even in case of different failures that may occur and are described further in this chapter. In order to improve reliability, the infrastructure optimizes the usage of available resources to provide correct stream processing results. In case correct stream processing is not possible, the infrastructure informs about the situation. This downtime decreases availability but reliability is maintained. Availability as another important requirement can be increased by offering more redundant components within the infrastructure.

3.2 Classification of Failures

3.2.1 Failures of Services

A common failure class is that an existing service, e.g., a running operator, is no longer available for process execution. This failure may be raised because of the following reasons:

- *Network failure*: Due to a network disconnection, the service is not reachable. This

failure is common in case of mobile devices with wireless network connection. Simply leaving the connection range causes such a failure. In this case the local HDB-Layer on the provider is working but not able to communicate.

- *Provider failure*: Due to a failure of a service provider, all services running on this provider are no longer available. This failure is common due to abnormal system ends of providers. In this case the local HDB-Layer on the provider has also failed.
- *Single service failure*: One service running on a service provider crashed. In this case the local HDB-Layer on the provider is working and able to communicate.

3.2.2 Failures of Resources

These failures are triggered if a service provider is not able to fulfill the requested task due to lack of resources. Load balancing allows to avoid and handle these classes of failures by smart distribution of processing tasks among the available provider nodes. The following resources may be subject for load balancing:

- *Memory overload*: Overflows of operator queues or memory intensive processing tasks cause this failure. Especially in case of mobile devices with limited memory, this failure is common.
- *Computation overload*: This failure is caused by computationally intensive processing tasks. For example, the service provider may be overloaded and is not able to continue the offer of computation resources. Again, in particular mobile devices are vulnerable to this failure case.
- *Network bandwidth overload*: High network load, produced by distributed data stream processing, can likely exceed the available network bandwidth. If network bandwidth overload leads to a complete disconnection, this failure class becomes a network failure.

3.2.3 Duration of Failures

Failures are also classified by the duration the failure exists. This classification is orthogonal to the previous failure classifications. According to the duration of a failure, the infrastructure has to treat the failure differently.

Temporary failures exist only for a short period of time, e.g., a short network disconnection or a crashed provider that reboots. *Permanent failures* exists for a longer period of time, e.g., the crashed provider is not able to reboot within the given timeout.

The tolerated timeout for a temporary failure depends on the characteristics of the affected process (e.g., the frequency of arriving data stream elements or defined quality of service parameters) but also on available resources for buffering. If the timeout is elapsed, the temporary failure becomes a permanent failure. Detection and handling of failures by our infrastructure is discussed in the next chapter.

4 Failure Handling for Data Stream Processing in OSIRIS-SE

Based on the background set in the previous sections, we now present the failure handling of OSIRIS-SE that allows to achieve a high degree of reliability and availability.

4.1 Failure detection of OSIRIS-SE

Detection of the applicable failure class is inevitable for proper failure handling. This detection is done by the HDB-Layer on each node of the infrastructure. Consequences of a failure usually affect more than one node of the infrastructure. Therefore, it is vital for proper failure handling, that all affected nodes detect the failure or are informed about the failure and in particular apply a coordinated failure handling strategy. In the following, we describe how our infrastructure is able to detect the described failure classes.

In order to detect a failure of service, the HDB-Layer on each provider has to observe other provider's services. Leveraging the nature of stream processing, combination of transmission and observation is reasonable. The infrastructure controls the transmission of data stream elements with an acknowledge protocol as described in [TS01]. Each operator receives acknowledge messages from downstream neighbors. Receiving no acknowledge messages is an indication for a failure of a service. Generally, the upstream operator is able to detect a failure of service but can not distinguish between a network failure, a provider failure, or a single service failure.

A failure of a resource is detected by the HDB-Layer on the affected provider node, which keeps track of available resources on its node. The current OSIRIS-SE infrastructure monitors the computation load of each provider and publishes this information via the load repository. Distinguishing between temporary and permanent failures is achieved by timeouts. If there is no recovery of the failure within the timeout, the infrastructure classifies the failure as a permanent failure.

4.2 Failure Handling in OSIRIS-SE

All kinds of temporary failures, e.g., a temporary network disconnection (loss of messages) or a temporary failure of service provider which recovers within the given time, are compensated by the output buffers of the upstream provider. After recovery, the upstream provider resents the data stream elements and receives an acknowledge message.

After exceeding the timeout, a temporary failure becomes a permanent failure. Permanent failures require a more sophisticated failure handling by the infrastructure. Both permanent failures of services and permanent failures of resources, require to migrate the operator instance with its aggregated operator state from the affected provider to another suitable provider. This task is called *operator migration*. In case of a failure of service, the operator migration is triggered by the HDB-Layer of the upstream provider. Since the

provider that hosted the operator is no longer reachable, the infrastructure needs a backup of the operator state to allow for correct operator migration. If there are multiple preceding operators (on different providers) in the process control flow (as for a join operator), the upstream provider responsible for operator routing is also responsible for operator migration similar to stream process activation. Other upstream providers are informed about the routing decision via subscriptions on the join repository. In case of a failure of resource, the HDB-Layer of the affected provider detects the failure and is able to perform the operator migration.

For permanent failures where no operator migration is possible, e.g., there is no suitable provider available, two additional failure handling mechanism may be applied by the infrastructure.

Alternative Processing Branches are defined to continue data stream processing. For example, patient Fred is leaving the connection range with his body sensors connected to his PDA. In the present case, the infrastructure layer on the PDA, as service provider, is not able to migrate the operators out of reach. Instead, the infrastructure layer activates an alternative processing branch starting from the operators running on the PDA. The alternative processing branch offers the use of alternative operators to aggregate and store data stream processing outcomes with less accuracy on the PDA itself during times of disconnections.

In cases where operator migration is not applicable and also no alternative processing branch is defined, *invocation of traditional processes* handle the failure. This traditional processes are defined by the process designer, which describe how to deal with the failure. The process user, e.g. Fred's physician, is able to customize this process. In case of Fred leaving the house, an alarm process can inform him to return in connection range or send an alarm SMS by using his cellular phone. Fred's cell phone number is a process parameter customized by his physician, who is the process user.

Compared to failure handling of traditional processes, stream processes need more sophisticated failure handling strategies. Due to the fact that operators of stream processes are continuously running, operator instances aggregate an operator state during their execution. Failure handling requires to migrate this running operator instance to a new provider. This migration requires to access a recent backup of the operator state for a correct initialization of the newly created operator instance.

4.3 Output Queue Trimming and Operator Backups

Each output stream of an operator has a corresponding output queue. The output queue contains outgoing data stream elements for one or more downstream neighbors. Our infrastructure allows for sending one output stream to multiple downstream neighbors. After sending, the sent elements still remain in the queue. Two kinds of acknowledge messages control the work and the trimming of the output queue. Acknowledge messages contain a timestamp in form of a sequence number to express, that all elements before this point in time are affected.

Firstly, a downstream neighbor sends a *receive-acknowledge* message, which indicates that all elements before the given timestamp have reached its input queue. This acknowledge only affects the transport mechanism, which means that the acknowledge elements are not send again in the next sending phase, but the downstream neighbor can still rely on this element and is able to request them again if necessary.

Secondly, the downstream operator sends a *trim-acknowledge* message, which indicates that all elements before the given timestamp have been processed and are no longer needed. If elements of the output queue are acknowledged in this way by all downstream operators the affected elements are removed from the output queue. The downstream neighbors are no longer able to request these elements again.

The consequences of the second acknowledge mechanism are highly relevant for failure handling and operator migration. If a running operator instance needs to be migrated to another provider, a new operator instance is created and initialized with the old operator state. This state corresponds to a distinct point in time, where certain elements of input streams have been processed. This processing has produced the restored processing state and the elements in the restored output queues. A new operator instance initialized with this operator state expects to process input elements from this distinct timestamp in order to seamlessly continue the output production. These necessary input elements are retrieved from the output queues of upstream operators.

If the infrastructure makes a reliable backup of an operator state, all input elements before this timestamp are no longer needed and acknowledged for trimming. In this case, side effects of data stream processing before this point of time are not produced again.

4.4 Unsynchronized Operator State Backups

In this section, we discuss how our infrastructure makes internal state backups and why the synchronization of backups is necessary for reliable data stream processing. In the simplest case, backups of operator states are not synchronized. *Unsynchronized Operator State Backups* allow for the infrastructure to decide for each operator a backup time without coordination with backups of other operators. For example, the infrastructure plans the backup in regular intervals individually for each operator.

For illustration, we consider a simple stream process consisting of a daisy chain of three operators (A, B, and C) as shown in Figure 4. Each operator backup is done by the infrastructure but not synchronized with other backups. If we look a the stream process at an arbitrary point in time the timestamps $(t_A^{out}, t_B^{in}, t_B^{out}, t_C^{in})$ of the operator state backups are in any order. Generally, a backup is referenced by one timestamp for each input stream, e.g., t_B^{in}, and each output stream, e.g., t_B^{out}, where the timestamp references the last element received or produced.

In case of a single failure, e.g., the provider hosting operator B fails, the provider of operator A detects the failure and is able to migrate the operator instance. A new operator B instance is started and preloaded with the operator state backup of timestamp (t_B^{in}, t_B^{out}). Since operator A knows that operator B has been migrated, operator A starts feeding op-

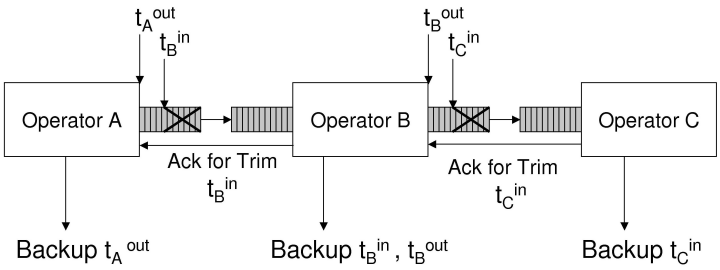

Figure 4: Unsynchronized Operator Backup

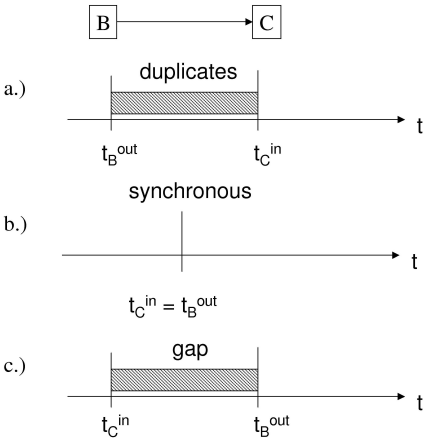

Figure 5: Times of backup for two consecutive operator migrations

erator B with data stream elements after the acknowledge for trimming timestamp which equals the backup timestamp t_B^{in}. The newly instantiated operator will start to produce data stream elements after the timestamp t_B^{out}. Since the failure has occurred some point in time before t_B^{in}, t_B^{out}, the new operator instance will produce some elements which the failed operator instance has already produced (all elements between t_B^{out} and the time of failure). For this reason, the input queue of operator C receives duplicates of some data stream elements, which are transparently dropped by the infrastructure. In case of single failure, unsynchronized operator state backups are sufficient for reliable data stream processing.

If two consecutive operators in the process control flow fail, unsynchronized operator backups perform different. For example, operator B and C will fail. We can define different cases dependent of backup times (see Fig. 5).

Firstly, the backup of operator B occurs before the backup of operator C ($t_B^{out} < t_C^{in}$) as illustrated in Fig. 5a. Operator B will be migrated by the infrastructure and restarts producing data stream elements at timestamp t_B^{out}. Operator C is also migrated and expects data stream elements after t_C^{in}. In this case, operator C will receive some duplicates of

tuples, which can be safely dropped.

Secondly, if the backups are synchronous ($t_B^{out} = t_C^{in}$) no duplicates are produced (see Fig. 5b). Generally, if backup times are ordered as in case a.) and b.), reliability for multiple failures is achieved, because not data elements are omitted from processing.

Thirdly, the backup of operator B is later than the backup of operator C ($t_B^{out} > t_C^{in}$), see Fig. 5c. If so, the new operator B produces elements after t_B^{out}. But the newly instantiated operator C expects to receive elements after t_C^{in}, which is later in time. In this case, unsynchronized operator backups are not able to correctly process an incoming stream and thus to provide reliable DSM. If quality of service definitions allow for limited reliability which tolerates small gaps in data streams, unsynchronized operator state backups are applicable.

In order to provide full reliability in case of multiple failures, our infrastructure has to guarantee that backups of preceding operators in the process control flow are always done later in time or synchronous. For this reason, we propose an algorithm in our infrastructure to achieve *synchronized operator backups*.

4.5 Synchronized Operator State Backups

Synchronized operator state backups describe an algorithm that applies to the planning of operator state backups in order to guarantee that the input backup timestamp of an operator is equal to the output backup timestamp of the preceding operator. This algorithm is a protocol between two subsequent operators in the process control flow and controls the times of operator backups. The protocol consists of two phases and the proceeding is similar to the two phase commit protocol [Gra78] ensuring global commits in distributed transactions.

In the first phase (*planning phase*), an operator backup is scheduled by the infrastructure. For unsynchronized backups, the infrastructure would make an operator state snapshot for permanent storage, but in the synchronized case the HDB-Layer stores the backup only temporarily not overwriting the former operator backup. The backup is planned for permanent storage, but the HDB-Layer has to guarantee that backups of subsequent operators in the process control flow match this backup. For this reason, backup-requests for the timestamp of the temporary backup are sent to all downstream neighbors. The output queue part in the operator backup contains only elements after the time of the backup request.

In the second phase (*backup phase*), the requests are executed by the HDB-Layer of all providers hosting downstream operators and appropriate acknowledge messages indicate the result of the backup request. Only if appropriate acknowledge messages of all downstream operators are received, the infrastructure overwrites the old operator backup with the temporary backup.

Revisiting our daisy chain of three operators again (see Fig. 4), we illustrate this algorithm according to Figure 6. In this scenario, our infrastructure provides synchronous backups of the three operators A,B, and C.

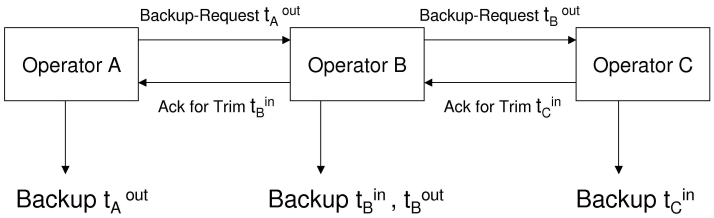

Figure 6: Synchronized Operator Backup

In the following, we describe how the infrastructure performs operator backup of A, which is synchronous to the backup of B. The HDB-Layer of operator A sends a backup request for timestamp t_A^{out} to the HDB-Layer of operator B. If the HDB-Layer of operator B sends an acknowledge indicating a trim for timestamp t_B^{in} and $t_A^{out} = t_B^{in}$, the HDB-Layer of operator A has an acknowledge of all downstream neighbors (operator B) and is allowed to save the backup of operator A permanently.

This protocol can be cascaded to chains of arbitrary length, because the infrastructure running the operator which receives the backup request may also start a synchronous backup by applying this protocol on the downstream neighbors. On the other hand, synchronized and unsynchronized operator backups can also be combined. For parts of the stream process where quality of service definitions require high reliability because even small gaps in data stream processing are not tolerable, synchronized backups are applied. For less critical parts of the stream process, unsynchronized backups are sufficient. Also splits in the process control flow are supported since requests are send to all downstream operators that follow the split and acknowledges of all downstream neighbors are necessary to perform the permanent backup. Additionally, network traffic overhead is reduced due to smaller output queues contained in the backups.

As for many tasks in data stream processing, we have to investigate the performance of synchronized operator backups on join operators more precisely. Since join operators have multiple preceding operators in the process control flow, they may receive backup requests on all inputs. Obeying all backup requests received from the inputs increases the frequency of backups. In particular, if synchronized backups are cascaded for the outputs, then also subsequent operators will suffer from this burden. For the reduction of this backup burden we will combine backup requests from different inputs by extending the existing operator backup with input elements of the time between the requests. These *extended synchronized operator backups* are applied if the backup requests are close in time. Close in time means that only a limited amount of input elements can be received between the backup requests. This limit is dynamically set by the infrastructure by comparing the sizes of extended operator backups with existing operator backups for the affected operator instance. An extended operator backup increases with the time elapsed since the last operator backup.

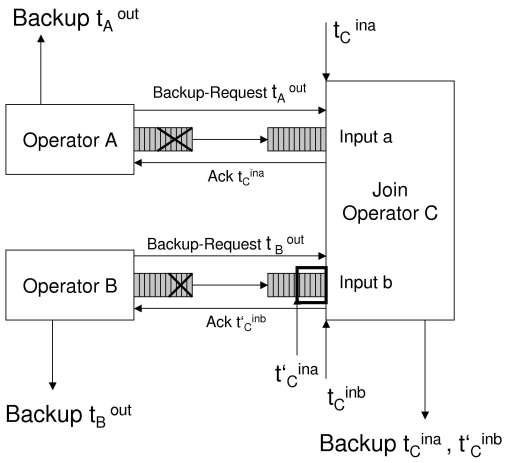

Figure 7: Extended Synchronized Operator Backup

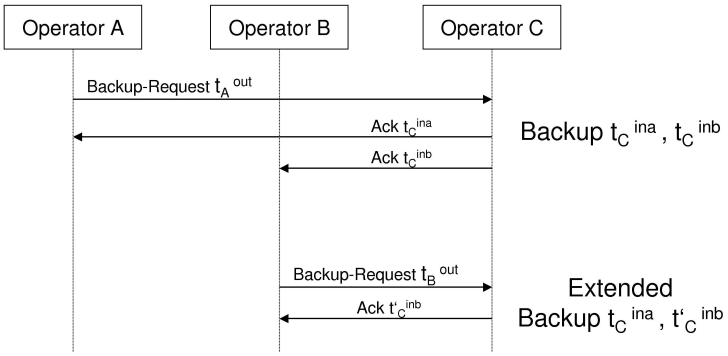

Figure 8: Extended Synchronized Operator Backup Sequence Diagram

4.6 Extended Synchronized Operator State Backups

Extended synchronized operator state backups allow for extending operator backups to the input side. This proceeding is useful if a join operator receives multiple operator requests that are close in time.

For illustration, we describe a simple stream process consisting of two operators (A and B) feeding a join operator C (see Fig. 7 and the sequence diagram in Fig. 8). Operator A sends a synchronous backup request at time t_A^{out}. Operator C receives this backup request on input a and initiates a backup referenced by t_C^{ina}, t_C^{inb}. Shortly later, operator C receives a backup request on input b for t_B^{out} from operator B. Shortly after means that operator C has processed some input elements on input b since the last backup ($t_B^{out} > t_C^{inb}$). For this reason, the backup of the subsequent operators C is prior to B. In case B and C fail and are recovered from their backups, operator C faces a gap in input b. In order to allow for

backup combination while at the same time avoiding the production of gaps, we introduce a mechanism to extend the existing backup. Revisiting our example scenario, operator C receives a backup request (t_B^{out}) on input b shortly after executing a backup referenced on this input by t_C^{inb}. Operator C extends the existing backup by the elements of input b between t_C^{inb} and t_b^{out}. Now, the current backup allows for acknowledging t_B^{out} and is referenced by $t_C^{ina}, t\text{`}_C^{inb}$ whereas $t\text{`}_C^{inb} = t_B^{out}$. Time references on other inputs or outputs are not affected by this extension.

The outcome of extended synchronized backups is even more beneficial if we think of cascaded synchronized operator updates. In this case, the extension affects only the pending temporary backup. Backup requests already sent to the downstream neighbors remain still valid for this backup.

5 Related Work

Data stream management has received an increasing popularity among researchers in the recent years. In what follows, we briefly introduce main DSM projects, focusing in particular on aspects of distribution and reliability.

Aurora [B$^+$04] allows for user defined continuous query processing by placing and connecting operators in a query plan. Queries are based on a set of well-defined operators. QoS definitions specify performance requirements. Extensions of Aurora also address DSM in distributed environments. In particular, in *Borealis* [B$^+$04], the extension of Aurora, the main emphasis is on aspects of distributed DSM. Algorithms for high available DSM in context of Aurora are discussed in [HBR$^+$03].

TelegraphCQ [C$^+$03] is a DSM project with special focus on adaptive query processing. *Fjords* allow for inter-module communication between an extensible set of operators enabling static and streaming data sources. *Eddies* describe adaptive query processing. Sets of operators are connected to the Eddy, and Eddy routes each tuple individually. *Flux* [SHCF03] provides load balancing and fault tolerance by providing adaptive partitioning of operator execution over multiple network nodes. This is realized by placing Flux between producer consumer pairs. Work on supporting high availability and fault-tolerance for Flux can be found in [SHB04].

The *StreamGlobe* [SK04] project aims at providing distributed DSM in heterogeneous peer-to-peer network environments. The infrastructure of StreamGlobe is based on the Grid reference implementation *Globus Toolkit* and an additional Peer-to-Peer network layer. Meta data distribution is based on *ObjectGlobe* [B$^+$01]. An additional aspect of this work are XML data streams and XQueries on these streams.

Pipes [KS04] offers a variety of basic building blocks for DSM in the Java XXL library. The library approach of Pipes enables the creation of a tailored DSM system to a specific application scenario. Pipes integrates a query construction framework and covers the functionality of the Continuous Query Language (CQL). Additional frameworks for scheduling, memory management, and query optimization are also provided.

The *P-Grid* [A+03] project is not directly related to DSM, but covers interesting issues that are also relevant to our infrastructure. P-Grid provides an advanced, fully decentralized peer-to-peer infrastructure. A peer-to-peer lookup system offers access to global available information replicated among the available peers. Reliability is achieved by redundant replication. Additionally, P-Grid also covers the problem of updates on these distributed replicas.

6 Conclusion

In this paper, we have introduced OSIRIS-SE, a stream-enabled hyperdatabase infrastructure that provides a high degree of reliability. For this reason, OSIRS-SE is particularly suited to be used in healthcare applications where a high degree of reliability is a vital requirement. Compared to other approaches in this field, our infrastructure offers some unique characteristics. Firstly, dynamic peer-to-peer process execution where reliable local execution is possible without centralized control. Secondly, the combination of DSM and process management in a single infrastructure offers new possibilities to process designers, like sophisticated customized failure handling, if recovery or operator migration is not available (e.g., definitions of alternative stream process branches or invocation of transactional processes for failure handling). Moreover, the OSIRIS-SE infrastructure supports extensibility, based on the integration of existing building blocks or new complex disease specific operators, as they can be found in healthcare applications, e.g., specialized processing of ECG data with side effects that store critical events in the patient record, or call an emergency service.

The OSIRIS-SE infrastructure is completed by an extended version of our process design tool O'Grape [WSN+03] that allows to define and deploy stream processes. Currently, OSIRIS-SE is used in comprehensive evaluations and experiments. First results (that can be found on our project-website [OSI]) measure the network traffic overhead of unsynchronized operator backups. These results indicate the high network traffic overhead this backup algorithm requires. More advanced evaluations will address the overhead of synchronized backups. Future work will concentrate on the support of mobile devices, especially on the dynamic re-routing of stream processes in case these devices will be disconnected.

References

[A+03] K. Aberer et al. Advanced Peer-to-Peer Networking: The P-Grid System and its Applications. *PIK Journal - Praxis der Informationsverarbeitung und Kommunikation, Special Issue on P2P Systems*, 2003.

[AHA01] American Heart Association. *2002 Heart and Stroke Statistical Update*, 2001.

[B+01] R. Braumandl et al. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[B+04] H. Balakrishnan et al. Retrospective on Aurora. *VLDB Journal*, 2004.

[BSS04] G. Brettlecker, H. Schuldt, and R. Schatz. Hyperdatabases for Peer–to–Peer Data Stream Processing. In *Proc. of ICWS Conf.*, pages 358–366, San Diego, CA, USA, 2004.

[C+03] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of CIDR Conf.*, Asilomar, CA, USA, 2003.

[Gra78] J. Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[HBR+03] J.H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Brown University, 2003.

[KS04] J. Krämer and B. Seeger. PIPES: A Public Infrastructure for Processing and Exploring streams. In *Proc. of ACM SIGMOD Conf.*, pages 925–926, 2004.

[OSI] The OSIRIS-SE project. http://ii.umit.at/osiris-se.

[PMJ02] S. Park, K. Mackenzie, and S. Jayaraman. The wearable motherboard: a framework for personalized mobile information processing (PMIP). In *Proc. of the 39th Conf. on Design Automation*, pages 170–174, 2002.

[SAS99] H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proc. of PODS Conf.*, pages 316–326, Philadelphia, PA, USA, 1999.

[SBG+00] H.-J. Schek, K. Böhm, T. Grabs, U. Röhm, H. Schuldt, and R. Weber. Hyperdatabases. In *Proc. of WISE Conf.*, pages 28–40, Hong Kong, China, 2000.

[SHB04] M.A. Shah, J.M. Hellerstein, and E. Brewer. High Available, Fault-Tolerant, Parallel Dataflows. In *Proc. of ACM SIGMOD Conf.*, pages 827–838, 2004.

[SHCF03] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proc. of ICDE Conf.*, Bangalore, India, 2003.

[SK04] B. Stegmaier and R. Kuntschke. StreamGlobe: Adaptive Anfragebearbeitung und Optimierung auf Datenströmen. In *Workshop Dynamische Informationsfusion, 34. Jahrestagung der Gesellschaft für Informatik*, 2004.

[SSSW02] H.-J. Schek, H. Schuldt, C. Schuler, and R. Weber. Infrastructure for Information Spaces. In *Proc. of ADBIS Conf.*, pages 22–36, Bratislava, Slovakia, 2002.

[SSW02] H.-J. Schek, H. Schuldt, and R. Weber. Hyperdatabases – Infrastructure for the Information Space. In *Proc. of VDB Conf.*, pages 1–15, Brisbane, Australia, 2002.

[SWSS03] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Peer–to–Peer Process Execution with OSIRIS. In *Proc. of ICSOC Conf.*, pages 483–498, Trento, Italy, 2003.

[SWSS04] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Scalable Peer–to–Peer Process Management – The OSIRIS Approach. In *Proc. of ICWS Conf.*, pages 26–34, San Diego, CA, USA, 2004.

[TS01] A. S. Tanenbaum and M. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[WSN+03] R. Weber, C. Schuler, P. Neukomm, H. Schuldt, and H.-J. Schek. Web Service Composition with OGrape and OSIRIS. In *Proc. of VLDB Conf.*, Berlin, Germany, 2003.