Safety Ring: Fault-tolerant Distributed Process Execution in OSIRIS

Nenad Stojnić Heiko Schuldt

Technical Report CS-2012-002
University of Basel
Email: {nenad.stojnic|heiko.schuldt}@unibas.ch

Abstract

The advent of service-oriented architectures (SOAs) has strongly facilitated the development and deployment of large-scale distributed (service-oriented) applications. The middleware for orchestrating process-based applications that consist of several distributed services has to be inherently distributed as well, in order to provide a high degree of scalability and to avoid a single point of failure. Self-healing execution of such processes supported by a distributed middleware requires replicated control metadata and instance data of processes. Most importantly, replication has to be provided in a way that does not affect the adaptivity and elasticity behavior of the middleware for composite service execution. In this technical report, we introduce *OSIRIS Safety Ring*, a novel approach to fault-tolerant process execution. Safety Ring is based on OSIRIS, a distributed and decentralized middleware for the execution of composite services. Essentially, the Safety Ring exploits dedicated node monitors, organized in a self-organizing ring structure, for the replication of control data. Moreover, it leverages virtual stable storage for managing process instance data in a robust way. We present the architecture of OSIRIS' Safety Ring and discuss in detail the algorithms it applies for self-healing process execution. The performance evaluation shows that the additional gain in robustness has only marginal effects on the scalability characteristics of the system.

Keywords:
Composite services; distributed process management; reliable distributed process execution; fault-tolerance; self-$*$ properties.

Department of Mathematics and Computer Science
Bernoullistrasse 16
Phone: +41 61 267 05 55 Fax: +41 61 267 05 59

Databases and Information Systems
CH – 4056 Basel, Switzerland.
URL: http://dbis.cs.unibas.ch/

1 Introduction

The huge success of service-oriented architectures has given rise to an important class of complex distributed applications that are built by combining existing services – this is also known as programming-in-the-large [DK76]. As such composite services, or (workflow) processes, are more and more used in domains where their reliable execution is crucial, they need to be executed in a robust and highly scalable way. This applies both for the processes and the middleware which orchestrates their execution. Hence, while processes are inherently distributed by nature, also the middleware supporting their execution has to be distributed as well, to avoid any single point of failure or potential performance bottleneck. However, guaranteeing a high degree of robustness and reliability for a distributed middleware poses new challenges, especially compared to traditional centralized middleware solutions, e.g., [CWEF06].

In this technical report, we introduce *OSIRIS Safety Ring* (Open Service Infrastructure for Reliable and Integrated process Support – Safety Ring), a middleware for the decentralized and distributed management of processes based on locally available metadata on process execution. This includes available services, instance data of processes, etc. Safety Ring is based on OSIRIS and adds support for the reliable execution of processes, even in environments where individual nodes might feature a high failure probability, for instance when incorporating mobile, resource-limited devices. Hence, Safety Ring has been built to combine both reliability and robustness, needed for business-critical applications [SST+05, STS+06] (e.g., for very large computational clusters or Cloud environments) with a small systems footprint, so that it can even be used with unreliable commodity hardware [BS11] (e.g., mobile devices and mobile service providers).

The distributed process execution follows a continuation passing style. Essentially, each service provider is equipped with a small middleware layer (called OSIRIS layer) that is in charge of locally invoking a service which is part of a process, and for managing the state of the process instance. After successful local service invocation, based on local metadata on the overall system (available services, their providers, their load, etc.), the middleware layer selects the most appropriate provider (e.g., the least loaded one) for the next services(s) to be invoked and passes control to the local middleware layer of the corresponding provider node(s). This is done until the execution of the complete process is successfully finished. The freshest metadata, that is locally needed for process instance routing, is provided by means of a sophisticated publish/subscribe (pub/sub) replication mechanism. According to this pub/sub mechanism, data is collected from all nodes and only the most necessary data is disseminated to nodes according to their subscriptions. This is done in the background and does not need to be updated at run-time, when a routing decision is to be made by a node. Moreover, a scalable key-value store is provided for the permanent storage of crucial metadata on nodes. Late binding of service instances, i.e., the selection of a provider for a particular service type at run-time by the predecessor of that service in the composition, in conjunction with load balancing strategies, guarantee that the workload is economically distributed across the available service instances – thus avoiding potential bottlenecks or single-points-of-failure and enabling the system to scale to a virtually unlimited number of nodes. However, failures of single nodes involved in the distributed execution of a process need to be handled correctly, in order to avoid that a complete composite process instance dies with the failure of a node that is currently in charge of executing it. In traditional approaches based on centralized workflow or process engines for composite services, this can be handled by means of persistently storing state information in a centralized instance database which is connected to several such engines (e.g., in the case of IBM WebSphere [KBG+10]). Yet, such existing centralized approaches to process

execution come with potential performance bottlenecks and single points of failure, and do not allow for advanced deployments, especially in mobile, resource-limited environments. Therefore, in this technical report, we address important aspects related to robust decentralized and distributed process execution, especially self-healing and self-adaptation properties.

1.1 Sample Use Cases

1.1.1 Mobile, Resource-Limited Devices

As an example for the use of OSIRIS-SR, consider a firefighter rescue scenario, e.g., the extinction of a fire where a group of firefighters is subdivided into smaller units and sent out into the field with preassigned specific tasks. In the course of an intervention, and induced by various environmental and human causes (e.g., panic, fear, fatigue), critical situations can occur that put the firefighters' lives into risk. For example, a sudden change of the wind direction can cause an unexpected movement of fire and thus entrap the firefighters and put them into danger. Due to the individual firemen's limited perception and view of the overall scene, they most probably cannot anticipate such situations. In order to reduce the risk, firefighters rely on mutual communication and coordination among each other and with the head quarters. However, the high dynamics of their tasks does not always allow them to communicate with the others. Moreover, for communication purposes, firefighters are usually only equipped with simple radio devices. Therefore, the coordination possibilities of the firefighters in the field are rather limited, and as a consequence the risks are kept high. However, their coordination effort could be facilitated by applying the latest technological advances in the fields of mobile communication, surveillance and intelligent decision support software.

As a vision for the future, consider a wearable intervention support system the firefighters will be equipped with. This system consists of a camera device, GPS sensor, accelerometer, and body sensors, integrated into a computation device for local processing, data storage, and wireless communication. Together, all these devices make allow to produce (record) data about the perception, the movement, the position of the individual firefighter in the course of an intervention. All the collected data can be wirelessly transmitted to the other firefighters' computation devices (reliability) and to a firefighter vehicle. The vehicle, in turn, can be equipped with more powerful sensor devices (environmental sensors, infrared cameras), which due to their weight and sensitivity cannot be carried by the firemen. Powerful base station computing devices can be located at the vehicles, as well. The data collected at the vehicle is shared with the other vehicles and is subject to sophisticated analyzes (possibly in real time). Based on the analyzed data, risks can be detected, strategies to prevent them devised (for future reference), and the firemen better coordinated.

However, the application of such a wearable intervention support system poses a certain amount of technical challenges. The most important aspects are resource provisioning and reliability. Essentially, the devices carried by the firefighters are, unlike the devices located on the vehicle, characterized by limited resources (e.g., CPU, battery, storage capacity, wireless reception), and the data produced at the various sources can differ in volume and thus in demand for resources than others. Consequently, the data should be shared with the other devices both for fault tolerance and resource utilization purposes. Therefore, the applicability of the firefighters' mobile devices are only of limited use and should be restricted to simple tasks (e.g., monitoring of other mobile devices, data filtering, data propagation), whereas the base station devices should be used for more resource demanding tasks (e.g., permanent storage, risk analysis). All the possible data

processing tasks can be encapsulated into services and distributed across different devices so that sophisticated applications consisting of several service invocations span multiple devices. Due to the inherent mobility of the users and their devices, it is rather likely that the execution of such applications is obstructed by failures. For example, it is reasonable to expect the devices, carried by the firefighters, to get damaged in the course of an intervention. Moreover, due to the movement of the firefighters, connection to other devices may get lost, or the transmission can be (temporarily) obstructed by interference or physical obstacles. Again, the limitation of available resources (e.g., device battery life) can be a further cause of failure at devices, as well. Induced by the failure of a device and its resulting loss of data, critical situations may possibly remain undetected by the decision support program and the danger for the firemen kept at a high level. To prevent the situation in which any crucial data are lost due to failure of mobile devices, enhancing the intervention support system with a sophisticated fault tolerance mechanism is of utmost importance. For this to work, the reliable intervention support system has to take into account the resource constraints of the devices in use. Thereby, reliability in the system should be achieved by exploiting redundancy on top of existing devices. For example, available redundant devices could be leveraged for data replication in a timely (e.g., before a device moves out of reception, or battery dies) and a resource friendly fashion that involves only the most necessary data. Additionally, all devices should be at any point in time subject to surveillance by other ones as to detect failures and even to predict them before they happen.

1.1.2 Services and Workflows in the Cloud

As a second example, consider a scenario which involves business processes in a Cloud environment. In such a scenario, companies offer (Web) services (e.g., for selling some product) via the Internet to customers for financial compensation. Under the hood, these Web services are implemented as rather complex business processes that involve multiple (partially ordered) activities, each again implemented as a service (e.g., checking the availability of a product in stock, shipping the product, etc.). To automate these processes, workflows are utilized which enable the execution of in the specified order, i.e., the invocation of the services implementing these activities. In order to run an actual workflow instance, a workflow engine is required to orchestrate the execution of the specified activities. Thereby, the orchestration proceeds by invoking the appropriate activity and by passing information (e.g., the results of a previous activity) from a completed activity to subsequent ones for further processing according to the given process specification. For the workflow engine to work properly, the knowledge about all existing processes in a system and all activities that are encompassed are necessary. Traditionally, centralized workflow engines are exploited for the orchestration of business processes (e.g., [KBG+10]).

However, being a source of income to the company, the availability of the Web services under consideration is essential at all times from an economic point of view. This also applies in the case of high load on the system or the failure of workflow-based process executions. For this reason, the Cloud which promises unlimited resources, has become increasingly popular. Induced by the transparent pricing of the Cloud vendors, the deployment of the workflow processes in the Cloud allows companies to run their services at acceptably low operational costs. Moreover, due to the sheer unlimited hardware resources of the Cloud, companies can accommodate, from a hardware point of view, for an almost unlimited number of customers (i.e., scalability, or, in Cloud terms, elasticity). However, the deployment of business processes and the workflow engines running them on the Cloud may cause problems, for instance when individual services fail or when the

workflow engines limit the degree of scalability that would be available from a hardware point of view. This usually requires costly manual intervention, especially for failure handling purposes. In general, as a process is typically composed of multiple constituent services, even the slightest unattended misbehavior of a single service can result in a total process failure. A misbehavior of an activity within a Cloud-based process can also be related to various Cloud software or hardware infrastructure faults. For example, no matter how good the Cloud software or hardware resources are maintained by the vendor, they may also break down (e.g., part of the Cloud center loses power).

To prevent situations in which business processes fail, Cloud customers are interested in highly scalable and at the same time reliable workflow engines that are impervious to the aforementioned problems of a Cloud deployment. In turn, Cloud customers need to make sure that their services are highly scalable and available. For the desired workflow engine it should be possible to instantly detect process activity failure, out of a potentially large number of concurrently running process instances, and to efficiently recover their execution for all processes. Thereby, the the remaining hardware resources should be used efficiently. To reliably persist any kind of data used by the processes, the workflow engine should comprise a scalable and and reliable data storage, as well – which implies that data storage has to be done in a redundant way, by leveraging replication mechanisms on top of distributed resources (data centers). On top of all, the workflow engine should remain lightweight. Finally, the prolonged execution times, induced by the reliable execution of the workflow engine, of the business processes should not affect the end user experience of the offered internet services. Hence, these requirements which include reliable resource usage while still being economic in resource utilization and the automated recovery from infrastructure-related failures, necessitate novel approaches to self-adaptation and self-healing in workflow engines on a wide range of computational devices (resource poor and rich, Cloud and non-Cloud).

1.2 The Safety Ring in a Nutshell

The continuation model of execution leveraged for distributed orchestration of individual process instances is resilient to a wide class of network and node failures. Both temporary and permanent node failures have to be overcome by means of timely replication of execution-related metadata to stable storage at several nodes in the system. Redundancy in the system prevents the loss of such metadata, and thus unnecessary repetition of possibly expensive computation. The volume of metadata to be replicated increases with the number of concurrently running process instances. Therefore, available resources need to be used efficiently. In particular, the replication degree has to be dynamically set so as to be able to handle different kinds of failures, while at the same time not allocating too much storage space for replication – this is especially important for resource-limited environments or for environments with nodes dynamically leaving or joining the system. Ideally, despite of replication mechanisms for providing fault tolerance and robustness, the system should scale with the number of nodes and concurrent process instances. In this technical report, we present an effective and scalable approach to self-healing data management within process instances that is able to correctly deal with a broad spectrum of possible failures.

The solution adopted is based on the concept of a scalable self-organizing "node monitor" overlay, called the *Safety Ring* in which every active node in the system is supervised by a dedicated monitor. As those supervisor nodes themselves may reside on unreliable hosts within the system, they are organized in a redundant way as well, and any node in the system can take over the role of the supervisor, i.e., this role is not exclusively predefined for a specific set of nodes.

However, only one instance in this set of redundant nodes, the leader, is at any point in time responsible for the actual supervision; if this responsible node fails, another leader is dynamically elected and the monitoring task is seamlessly continued.

1.3 Structure

The remainder of this technical report is organized as follows. Section 2 introduces the OSIRIS-SR system model and basic notions regarding processes execution in OSIRIS. Section 3 presents in detail the Safety Ring, our solution for scalable and flexible decentralized failure handling. A quantitative and qualitative evaluation of our approach is given in Section 4. Section 5 presents related work and Section 6 concludes.

2 System Model

Safety Ring is an extension of OSIRIS (Open Service Infrastructure for Reliable and Integrated process Support) [SST+05, STS+06] that, among other features, provides sophisticated self-healing and self-adaptation capabilities for a scalable and efficient peer-to-peer based distributed and decentralized process execution. Thereby, an OSIRIS process can be considered as a program that specifies the invocation of distributed (Web) services. As these distributed services reside in the peer-to-peer environment, the peers that offer (i.e., host) them become service providers for an executing process. The decentralized process execution of OSIRIS takes into account dynamics that large-scale distributed environments can exhibit. Among others, it handles the following dynamics:

- Service providers may enter or leave the network.
- New services can be offered, existing ones may be withdrawn.
- Existing services may fail.
- New processes can be defined, old ones deleted.
- The load of the service providers is changing continuously.

Effectiveness and scalability of the OSIRIS approach is guaranteed by the idea to perform the execution of processes at peers based on only locally available and freshest metadata about the system. In turn, global information about the system, possibly unnecessary for the execution of processes, is maintained at global metadata repositories and partially disseminated to peers, participating to process execution, only when needed. Thereby, the propagation of the metadata in a pub/sub style is completely decoupled from the process execution, i.e., peers always have at run-time sufficient information to manage processes.

To run the decentralized process execution, all of the necessary data management and process execution responsibilities are encapsulated into several roles and distributed among all the peers in the environment. Even though each peer is potentially able to take over all relevant tasks, there is no “super” peer that is explicitly chosen to be always in charge of all (most) of these tasks. Each peer can contribute to process support, according to its locally available resource capacities, by taking over one or several roles:

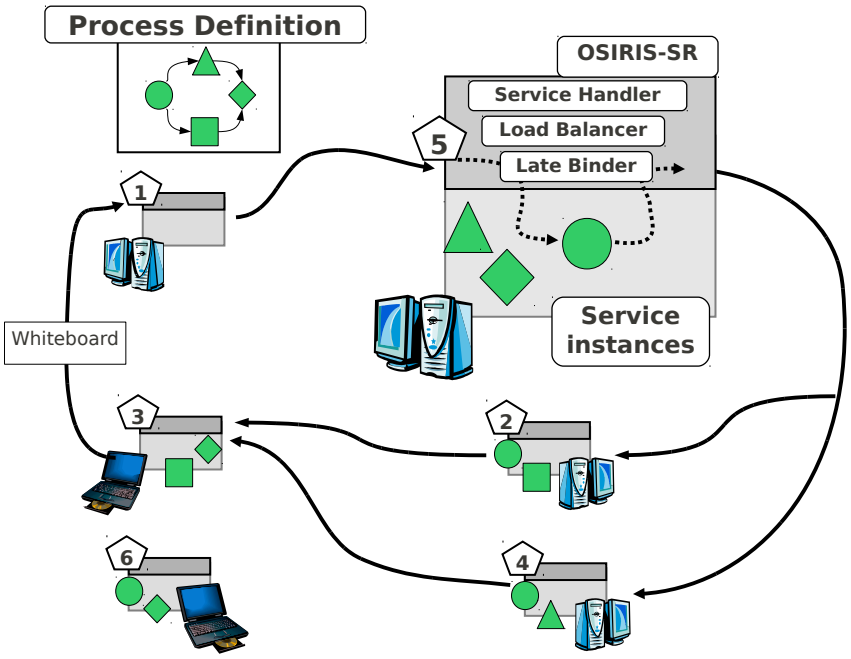

Figure 1: Distributed Process Execution in OSIRIS

1. A peer can be a provider of a service that it locally hosts, i.e., a *service host*;
2. It can be an OSIRIS process execution support node, i.e., a "*worker*", in which case it provides local functionality for process instance invocation, navigation, and routing.;
3. A peer can also be a dedicated node fault handler, i.e., a *SR-node (Safety Ring node)*, in which case it provides local functionality for peer monitoring, data consistency enforcement, ring topology construction, and failure recovery.;
4. Finally, a peer can be a data management node, i.e., a *SM-node (shared memory node)*, in which case it provides local functionality for reliable storage and dissemination of global execution related data, essential for the other roles to function.

By default, any service provider that wants to participate to an OSIRIS cluster is at start-up equipped with the data management role as well as the process support role. Whether a node additionally takes over the fault-handling role is randomly determined at start-up of the node, also based on the already existing Safety Ring nodes.

From an architectural point of view, all the aforementioned roles and their respective functionalities are implemented within three lightweight software layers (c.f. Figure 2) that can be deployed in any combination, at any peer in the network. The first layer, the core process support layer (Figure 2.a), corresponds to the worker role. The second layer, the Shared Memory layer (Figure 2.b), corresponds to the data management role. The third layer, the Safety Ring layer (Figure 2.c), corresponds to the fault handler.

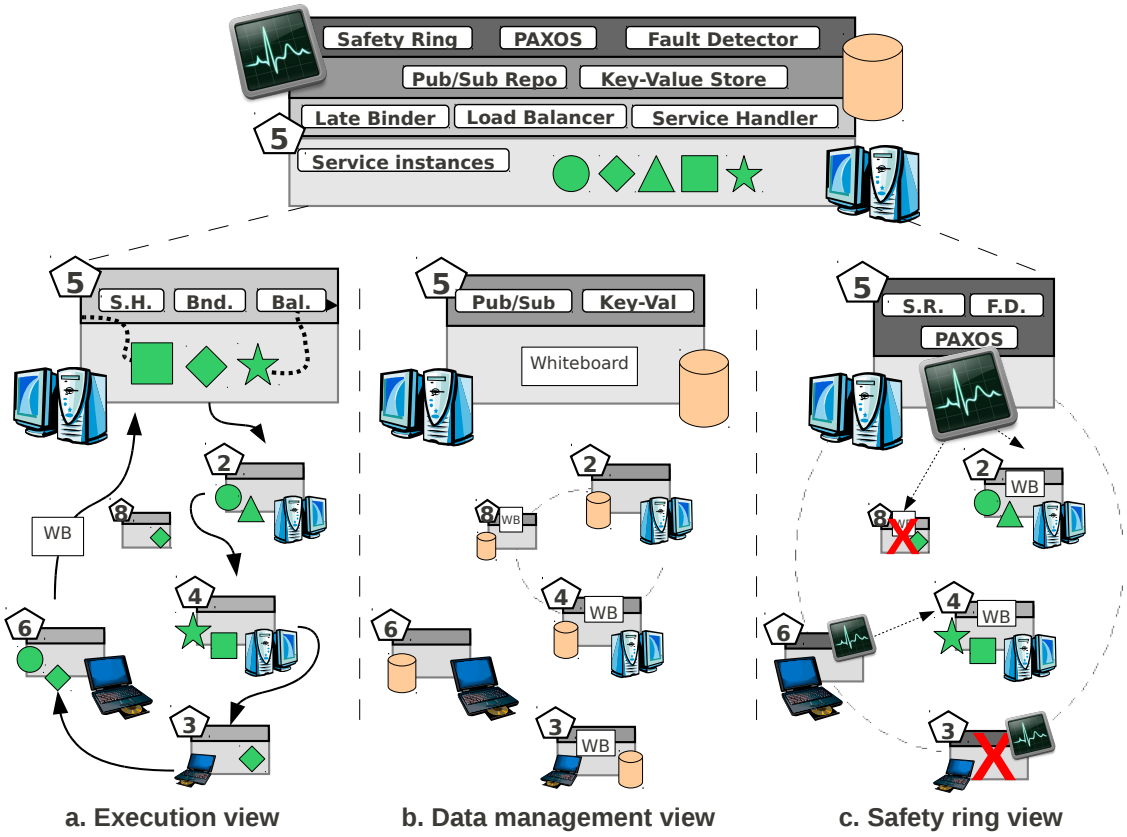

Figure 2: OSRIS-SR: Different Node Functionalities and Architectural Layers

Finally, with respect to the well established distributed environment abstractions we consider the OSIRIS setting to be characterized by the partially synchronous communication model and the fail-noisy fault model [CGR11]. In this abstraction, communication channels are reliable channels, processes crash with a crash-stop behavior, and an eventually perfect failure detector [CGR11] is available at any node (SR-nodes in our case). In other words, if no process at either end of a channel crashes, messages are eventually reliably delivered and delivery is acknowledged. An eventually perfect failure detector (installed at every SR-node) allows nodes to detect failure of other nodes (workers). This failure detector assumes nodes that do not respond to regular heartbeat messages to have crashed. In case of network congestions, and thus resulting delays in response time, this detector revises its failure assumption by taking the resulting delay into account when the monitored node eventually responds to the heartbeat messages. Additionally, we assume that all services invoked by worker nodes are fail-safe. By this we mean that a service that fails does not leave behind any uncompensated side-effect. Other important topics, such as the security of the communicated data, fail-unsafe services are outside of the scope of this technical report, however they might become subject to future research.

2.1 Distributed Process Execution

An OSIRIS-SR process description encompasses an ordered set of activities, each of them corresponding to the invocation of a (Web) service (either atomic or again composite), and describes the control and data flow between them. Service descriptions are abstract, i.e., they only specify the type or class of services to be invoked. The actual binding of services to concrete service instances is decided entirely by the local OSIRIS middleware at run-time (late binding), depending on the current configuration of the system (availability, load, cost of invocation). In doing so, local service instance providers are favored over remote service providers as such late binding results in less network traffic and better local resource leverage. Following the paradigm of service-oriented architectures [PH07], processes themselves are wrapped by service interfaces, and can be invoked from within other services. The data flow is defined as a sequence of mappings from a data space into the process instance, called the *whiteboard*, to the service request parameters and back. Hence, the whiteboard holds all information about the activity and the local service call.

The continuation model based execution proceeds in a purely decentralized peer-to-peer fashion that involves only those nodes that offer a service required by the process definition. For this to work, OSIRIS requires some nodes (i.e., some SM-nodes) in the environment to serve as system set-up information repositories. These system repositories maintain information on all available service providers (e.g., thier load and address), all available services and all existing processes. By default, each OSIRIS node is required to publish its local state in terms of hosted services, current load and running processes to these repositories and in turn is expected to receive information about other nodes from them when needed for the purpose of executing a process. For example, when a new process definition enters the system, its direct service dependencies (e.g., a service invocation of type B depends on — directly follows — the service invocation of type A in process P) get extracted and all nodes hosting a service of type A which may precede an invocation of service type B get updated by the repositories with data about all other nodes hosting the other dependent service. In general, the decentralized process execution in OSIRIS is by default backed by three global metadata repositories, namely the service repository, the process repository and the load repository, and any OSIRIS worker node is always in interaction with all of them. The service repository manages lists on all service hosts in the system and their addresses. The load repository keeps track of the dynamic configuration and workload of the system. Finally, the process repository, holds the global definitions of all available process instances. Note that all these repositories do not require a centralized implementation, but span multiple SM-nodes by using a pub/sub mechanism that is elaborated in Section 2.2.

At runtime, a worker selects a service host based on costs, parameters, and conditions on its services. The relevant data, needed to locally decide on subsequent service invocations is retrieved from the repositories residing at the SM-nodes before the actual invocation of the service happens. Upon completion of a service invocation, the control over the execution migrates to one or more successor nodes, by delivering a migration token containing flow-control information and the whiteboard. To ensure that the data transfer between two service providers is properly executed, transactional guarantees are enforced. In order to participate to distributed process execution, providers deploy the process support software layer (worker role) as depicted in Figure 2.a.

A simplified example of a process execution is shown in Figure 1. The figure illustrates a set of network nodes (numbered 1 to 6), each equipped with an OSIRIS middleware layer and each hosting a set of service instances (△, □, ◯, ◊) of different type (each shape denotes a different service type), some of them hosting metadata repositories. The process definition at the top of

Figure 1 specifies a sequence of four service invocations. Note that two services, △ and □, can be invoked in parallel as they are not ordered. The configuration of the system is as follows:

- All nodes are workers.
- All nodes except for nodes #1 and #7 are service hosts.
- Node #2 hosts the services of type ◯ and □.
- Node #3 hosts the services of type ◊ and □.
- Node #4 hosts the services of type ◯ and △.
- Node #5 hosts the services of type ◯ and □.
- Node #6 hosts the services of type ◯ and ◊. It is also a SM-node and it hosts the process description repository.
- Node #7 is also a SM-node and it hosts the service repository and the load repository.

The sample execution proceeds as follows:

1. Assume that the execution is started at node #1 by receiving the process description. The OSIRIS layer on node #1 compiles the process description by decomposing it into the individual service invocation steps and uploads it to a process description repository at node #6.
2. Since all workers are always subscribed to the process description repository, all of them learn about the new process description. Depending on the locally hosted services, workers subscribe at the service repository residing at node#7 for other service hosts and their respective loads according to the process definition dependencies. In other words, since node #5 hosts the service ◯ and this service is followed by services △ and □, node #5 subscribes at the service repository for the host addresses and their load of △ and □. Moreover, it subscribes to ◊ as it follows the □ service. Node #2 subscribes as node #5 does (hosted services are the same). Node #4 subscribes to △, □ and ◊. Node #3 subscribes to ◊ despite hosting it (as it also hosts □ which is a potential predecessor). Node #6 subscribes to △, □.
3. Once all subscriptions have been made, node #7 pushes to the subscribed workers information of interest. Node #5 receives the address and the load information about node #4 (host of △), node #3 (host of □ and ◊), node #6 (host of ◊) and node #2 (host of □). Node #2 receives the same data as node #5 does. Node #3 receives information about node #6 (host of ◊). Node #4 receives information about node #3 (host of □ and ◊), node #5 (host of ◊) and nodes #5 and #2 (hosts of □). Finally, node #6 receives information about node #4 (host of △) and the nodes #2, #5 and #3 (hosts of □).
4. Now, that the replication of the locally needed metadata has taken place, node #1 can initiate the process execution. According to the process description, the OSIRIS layer on that node must route the process instance to a service host of type ◯. This is the case for nodes #2 and #5. The actual node to which the execution is forwarded is determined at run-time (late binding) based on (received) local load information. In our example, we assume that node #5 has been chosen.

5. Node #1 thus migrates the execution to node #5, exchanging with it a migration token that contains control information and the whiteboard.

6. On node #5, the OSIRIS layer invokes the ◯ service. After the service instance has successfully finished its execution, the OSIRIS layer of the node must again pick a provider able to execute the next activity and route the execution to it.

7. In our example, there are two services that are to be invoked in parallel after ◯, therefore execution migrates from node #5 to node #4 (invoking a service of type △) and to node #2 (invoking a service of type □).

8. After both parallel service invocations have been completed, their respective invocation results (whiteboards) need to be joined on one provider capable of continuing the execution. In our case, the two execution threads on node #4 and node #2 join on node #3. The actual decision on which node to join will be explained in more detail in Section 3.

9. The OSIRIS layer at the node executing the last activity ◊ (node #3) will transfer the results to the invoking node of the process (node #1).

2.2 Distributed Data Management

The decentralized execution and orchestration of processes is backed by locally available system configuration related metadata at workers, obtained form the global repositories (load, service and process). This metadata includes information used for routing (e.g., addresses of other nodes, hosted services at other nodes), load balancing (workload of other nodes) and process instance backups (node activity results). In order to supply the workers with the needed metadata, OSIRIS relies on SM-nodes. These SM-nodes serve as global metadata repositories by collecting and storing data on the whole system and propagating it when needed. The actual propagation of data performed by the SM-nodes is based on two mechanisms. One is a pub/sub repository and the other is a key-value store.

Workers that actually execute process instances should not have to query for the metadata from the SM-nodes if the execution is to stay effective. Rather, a push mechanism should replicate data towards the workers even before the actual execution takes place (pre-instantiation). To qualify for pre-execution data propagation (publication), workers subscribe at SM-nodes for the data items they are interested in, i.e., data that they need in order to be able to navigate to the subsequent step of the process execution. Changes of the global metadata at the SM-nodes are propagated (published) only to the workers that have subscribed for it. Moreover, the changes on the global data can only be inflicted by the subscribed peers, and learned by the rest. At SM-nodes, the actual metadata is stored in a semi-structured representation (XML documents). To reduce the amount of replicated data, each subscription comprises a path predicate (XPath expression) which makes the peer subscribe only for a part of the whole XML document. To avoid unnecessary updates, each subscription further comprises a freshness predicate (eager, lazy) that defines when data items are to be propagated. An "eager" freshness predicate, for instance, instructs the repository to immediately publish changes to all nodes of interest. Moreover, freshness predicates help to reduce the amount of replicated data, especially in the case of highly dynamic data such as load. For example, specified by the freshness predicate, the repository may be instructed to update the subscribed peers only if,for instance, the load of a peer changes by more than 10%. In case of a

change due to an insertion, update, or deletion operation of some data, item all nodes that are affected by this change are determined via the subscriptions in the repository and are informed about the change. For example, when a service provider is about to leave the system, the service repository informs all nodes that are subscribed to this service provider, due to a process dependency, and all nodes can safely discard their local knowledge on the leaving service provider. Further details and example cases on the the pub/sub mechanism can be found in [SST+05, STS+06].

However, constant and global replication of data, for the purpose of pre-instantiation, may prove to be too costly in terms of resource consumption (e.g., bandwidth, CPU, storage), especially in the case of resource-limited (mobile) devices, and is better suited for large computational clusters where resources are not an issue. Moreover, crucial process execution data, such as service invocation results (the process instance's whiteboard), usually spans multiple workers during the course of an execution and it is absolutely necessary for this kind of data be always accessible to all nodes, even in the event of the failure of an active worker node, if execution is to function properly. Even more, as with the increase of concurrently running process instances, also the volume of instance data increases that will have to be accessed efficiently by any node. Therefore, means of accommodating for the growth of reliable data have to be provided. For this, OSIRIS provides a reliable and scalable stable storage in the form of a key-value store. Our implementation leverages the Chord [SMK+01] Distributed Hash Table (DHT). Since data persisted in the key-value store is guaranteed to be accessible as long as there are enough functioning nodes to host them, every SM-node is additionally enriched with key-value data storage functionalities. In other words, every SM-node also participates to the construction of the Chord DHT key-value store. In a key-value store, each data item is residing at a (usually different) small set of SM-nodes, defined by the replication factor, and is thus available to workers for retrieval. More concretely, workers can query the highly accessible and scalable key-value store for data at the same time as invoking a service instance, and thus still remain effective.

2.3 Self-healing Distributed Execution

The OSIRIS approach does not constrain the deployment of service instances in a network of nodes and allows any node that is equipped with the OSIRIS middleware services to orchestrate (parts of) a process execution. Although this approach is quite powerful and allows for a high degree of scalability [STS+06], it may also lead to situations in which the execution can be obstructed due to the failure of nodes participating to the execution. Especially in heterogeneous node environments, node failures, deriving from various causes (hardware breakdowns, network link failures, software errors) have to be taken into account. Therefore, we introduce a self-healing mechanism, called the Safety Ring, that detects node breakdowns and conducts appropriate recovery measures in a way that is scalable, reliable, and independent of the specific nodes involved in execution.

The Safety Ring (see Figure 2.c) mechanism is based on the idea of making each active worker node (service provider) subject to supervision by one SR-node. The responsible SR-node, in turn, is charge of worker node monitoring and failure recovery. For fault detection purposes, each SR-node is by default equipped with an eventually perfect failure detector [CGR11], that inspects the monitored workers for liveliness by means of heartbeat messages. In the event of a node failure, the designated responsible SR-node elects a replacement service providing node of the same type by means of late binding and thus recovers the execution of the failed process. Upon selection, the replacement node is provided with the same whiteboard (retrieved from the reliable key-value store) as the crashed node was, and the activity during which the failure occurred is restarted.

Finally, any intermediate results produced in the whiteboard by the crashed node is discarded from the key-value store.

To guarantee continuous and uninterrupted supervision of workers even in the presence of SR-node failures, supervision responsibilities are shared with other SR-nodes. The dynamically determined set of SR-nodes among which the state of the monitored worker nodes along with execution control data is shared is called the *Replica Pool*. A crashed pool member is detected by another SR-node, in the exact same way as worker crashes are detected, as every SR-node is assigned to one other distinct SR-node for monitoring. Upon failure detection, the crash detecting SR-node acquires the position of the crashed SR-node in the pool and thus "refills" it again. In this way, the replica pool is always kept intact and resilient to future crashes of other SR-nodes.

Scalable node clustering for any kind of interaction (e.g., monitoring) is achieved by means of node ring topologies. In such settings, each node is agnostic to the full set of nodes in the environment, but is only aware of a small (and ordered) subset of selected nodes, which allows for overall effective node interaction. This set includes the neighboring nodes (only two) in the ring plus a few $log(N)$ other nodes, in which N corresponds to the total number of all nodes. The details on how a ring is constructed and the neighboring node sets are determined is given in Section 3.

In order to ensure consistency of all data (whiteboard, execution control data, etc.) shared by multiple nodes (e.g., SM-nodes) a novel migration algorithm is provided, that is based on the Paxos Commit transactional protocol [SRHS10a]. Paxos Commit is a non-blocking protocol, as it is resilient to specific case of the transactional manager failure [SRHS10a], which gives our migration algorithm a high level of robustness. Our new migration algorithm replicates the whiteboard along to the worker, responsible of continuing the execution, also to the replica pool (SR-nodes). By additionally replicating to the pool, we inform the pool members that migration has taken place, and that a new worker is subject to monitoring.

The example illustrated in Figure 1 so far assumes an execution without failures. However, the continuation passing style execution may be affected by the following types of failures: state migration failure, preceding node failure, succeeding node failure, and active node failure. Revisiting the sample execution illustrated in Figure 1, these failures can be addressed as follows:

- *Migration Failure*. If during the forwarding of the whiteboard between any two service providing nodes a failure occurs (e.g., migration of control from node #1 to node #5 in step #4), the Paxos transactional protocol will cancel the migration and the remaining node will stay in a consistent state.

- *Preceding Node Failure*. If a node has crashed that has already finished its activity and that has already migrated the data state, no action is necessary as the crashed node was only responsible for the finished activity and has already forwarded control).

- *Active Node Failure*. If a provider that is actively involved in the execution of a process instance fails temporarily (e.g., node #5 when invoking service ◯), no handling is necessary, as data is locally preserved on stable storage. However, if this node fails permanently and the data located in the stable storage becomes unaccessible, the Safety Ring mechanism will take action in making the SR-node responsible of monitoring the failed node find a suitable replacement node (e.g., node #2 for step #6) and providing it with the same whiteboard, retrieved from some SM-node (key-value store), as the failed node.

- *Succeeding Node Failure*. If a node that is chosen by the currently active provider to continue the execution crashes, and if this crash happens before the migration itself, the active provider is capable of choosing a different node for migration (late binding). However, if the failed succeeding node is a join node, i.e., a node on which multiple parallel execution branches have to join, such as node #3 in our example, the Safety Ring has to take action. The failure of such a node, even of temporary nature, could result in incorrect execution. The reason is that because of join node failure timing, some nodes might decide (late binding) to join on the recovered failed node while others might decide to join on some replacement join node, causing an execution deadlock. Therefore, the responsibility of merging the branches is delegated to the SR-nodes which are resilient to failures. The SR-node responsible of monitoring parallel workers will continue the execution upon receiving all whiteboards by migrating a merged whiteboard to the next node.

Finally, for the proposed self-healing execution to fully function, we assume that there are always enough substitution service providers to choose from in the event of worker node failure. Analogously, we also assume that there are always enough substitution SR-nodes in the system to replace the failed ones, i.e., to "refill" the pools and to resume monitoring. The problems of having insufficient service providers and SR-nodes are outside of the scope of this technical report. However they could be overcome with dynamic service deployment in the former case and by enabling SR-nodes to promote regular workers to SR-nodes at will in the latter case.

3 Safety Ring

OSIRIS aims at a large spectrum of possible deployments – from a small number of powerful servers up to a very large number of resource-limited mobile and heterogeneous devices. In all these cases, workers require reliable supervision by SR-nodes. Therefore, effective means of distributing the workers among the SR-nodes for supervision in highly dynamic and large environments have to be provided. Even though each active worker node is monitored at any point in time by one SR-node, additional SR-nodes have to be available to take over in case the monitoring node fails. The distribution of SR-nodes has to take into account the nodes' load. The failure of an SR-node has to be detected immediately by other SR-nodes and should result in an effective reassignment to another SR-node. These properties can effectively and efficiently be provided by means of DHT-based node ring topologies.

The construction of a ring topology necessitates a unique circular position for each node, which is achieved by a consistent hashing function (*SHA1*) [SMK+01]. SHA1 is responsible for the unique mapping of some physical node identifier, such as the IP address, into a circular space. Given the unique circular space mapping, any node taking over the role of a worker will be at any point in time be located between two nodes currently taking over the roles of SR-nodes. Therefore, OSIRIS assigns in its Safety Ring workers lying between two SR-nodes in the circular identifier space to a SR-node for supervision by always selecting as monitor the SR-node with the higher circular ID for this purpose.

Since workers are only temporarily associated with SR-nodes (i.e., only while they are executing a service), making them participate to the ring topology construction would result in too many changes of the ring topology (high churn rates). Hence, the Safety Ring is only built out of SR-nodes. In order to construct the ring, each SR-node will link-up with only one succeeding node and with one preceding node in the circular identifier space, thereby constructing the

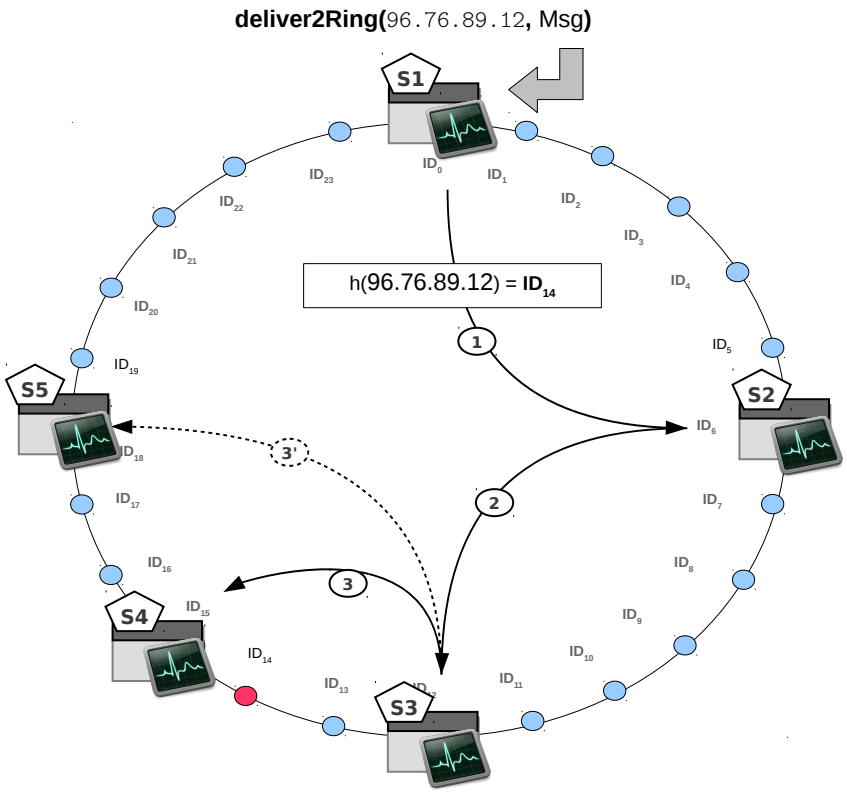

Figure 3: Node Ring Topology

ring topology. By periodically exchanging information about about their immediate neighbors (predecessor and successor) with other nodes, by means of gossip based protocols, each SR-node eventually learns about other SR-node joining or leaving the ring. For the sake of scalability, this accumulated neighbors list will not exceed the value of $log(SR)$ with SR being the total number of SR-nodes. More details on how Chord rings are constructed and maintained can be found in [SMK+01].

Figure 3 illustrates an example Safety Ring and the positioning of nodes in the ring space, in which case the smaller blue circles correspond to worker nodes, the bigger gray rectangles to SR-nodes, and the ID node labels to their respective circular identifier. For example, the worker with the circular ID_{14} (colored in red) is located between SR-nodes with the identifiers ID_{15} and ID_{12}, with the one with higher one (ID_{15}) being the responsible SR-node.

Even though they are only logically a member of the ring topology, worker nodes can nevertheless reliably interact with their SR-node by means of the `deliver2Ring(sndIP, Message)` primitive. `sndIP` identifies a worker node, and `Message` the data that the worker wants to communicate to its responsible node. By applying this primitive at any SR-node, the communicated message is forwarded in the ring topology by other SR-nodes until it reaches the responsible SR-node for the worker node identifier associated with the message. Algorithm 1 illustrates the functionality of the `deliver2Ring(sndIP, Message)` primitive. The communication in the ring relies on an efficient routing algorithm (Chord [SMK+01]), that guarantees $O(log(N))$ hops before reaching the node responsible for an identifier. Figure 3 depicts how by apply-

ing `deliver2Ring(sndIP, Message)` at some SR-node (S_1) a message originated form a worker with the IP address `96.76.89.12` is forwarded (following the solid arrows labeled 1, 2 and 3) among the SR-nodes (S_2 and S_3) until it reaches the responsible SR-node (S_4). If a SR-node for any reason (e.g., failure) leaves the ring, it will be simply detected and substituted by the succeeding SR-node in the ring, and the responsibility for workers reassigned accordingly. For example, if in Figure 3 the SR-node S_4 crashes, the SR-node S_5 becomes the next SR-node with higher ID for our worker identified by ID_{14} (red circle) and thus will be responsible for it. As a consequence, all messages communicated from this worker (ID_{14}) by means of `deliver2Ring(sndIP, Message)` will now be forwarded to S_5 (dashed arrow numbered with 3') by S_3. By continuously propagating their neighbor lists to others, all SR-nodes eventually learn about changes in the ring topology, such as SR-nodes joining or leaving. Moreover, for performance reasons, the Safety Ring features the primitive `deliverDirectly(destIP, Message)` as well, which allows workers to communicate messages to destination nodes directly (now, `destIP` corresponds to the identifier of the destination node), without having to reliably route the messages through the SR-nodes.

Algorithm 1: `deliver2Ring(sndIP, Message)`

Data: $sndIP$ identifier of the sender and Message to be delivered

```
1  begin
2      senderID ← hash(sndIP)
3      ownID ← getOwnID()
4      predID ← getPredecessorID()
5      if predID < senderID and senderID ≤ ownID then
6          handle(Message)
7      else
8          node ← routeToClosestPrecedingNode(senderID)
9          node.deliver2Ring(sndIP, Message)
10     end
11 end
```

Note, that resorting to a ring structure for node assignment comes with the price of having uneven ring chunk distribution among the nodes, churn, byzantine nodes, etc. However, those problems are outside of the scope of this technical report, and can be overcome by applying solutions such as provided in [MR08, RGRK04].

3.1 Replica Pool

The Safety Ring introduced so far only partly addresses SR-node failures. In order to successfully recover from worker node failures, a SR-node has to keep a certain amount of process execution related metadata on its subjected service providers. This service provider metadata includes important information such as current process instance and activity that the service provider is servicing and the activity input whiteboard. When a worker fails, this is the data that has to be replayed by the supervising SR-node to a replacement service provider if self-healing execution is to function properly. For the purpose of safeguarding this data even from SR-node crashes, the Safety Ring mechanism chooses to replicate this metadata in its peer-to-peer overlay. Therefore, we introduce a set of service provider metadata sharing nodes that we name the the *Replica Pool*. Although service provider metadata resides within the pool on multiple nodes, it is not desired that

all of them react to the changes of this metadata at the same time, as this kind of behavior could result in redundant failure handling. For example, consider again Figure 3, when a worker crashes during the execution of an activity (e.g., node #5 with the ID_{14} while servicing ◯) only its direct responsible (S_4) has to react to it (according to the unique Safety Ring assignment) by

i.) replaying the whiteboard to a replacement service provider of the same type (e.g., worker with the ID_{13}, offering the service ◯ as well)

ii.) triggering its fault detector for monitoring of the replacement node (again, worker with the ID_{13}), and

iii.) updating the metadata of the replacement and failed service providers in the pool.

Although maintaining metadata on the failed service provider, it is absolutely unnecessary for the other failed service pool members to perform the same actions as the direct responsible. They only need to acknowledge the change of the active service provider metadata induced by fault handling pool member. Therefore, we dynamically designate each pool with a leader that is in charge of updating the metadata and conducting the actions associated with it. The task of the other pool members is to keep the metadata redundant and to react to pool leader crashes, in sharing it with the replacement pool member. In detail, any failed pool member is replaced by the SR-node succeeding it in the Safety Ring and the lost metadata is retrieved from the rest of the pool, in which only the new leader will resume the activities (e.g., monitoring of service providers) associated with the obtained metadata.

In order to dynamically determine the pool members we apply symmetric replication [GAH07]. The set of metadata sharing nodes is implicitly determined by applying (1) where ID_x corresponds to the circular identifier of the data object to be replicated, and ID_i corresponds to the computed circular identifier of the replicated destination object. *N* is the circular identifier space size, *R* to the desired replication factor, and *i* to the replication iterator. The actual node responsible for the physical storage of the data object is thus found in the Safety Ring by means of the `deliver2Ring(`ID_i`, Message)` primitive, in which case ID_i corresponds to the newly computed symmetric, circular identifier and the replicated object to be stored to `Message`. The nodes responsible for the computed symmetric circular identifiers thus form the Replication Pool, and the one responsible for identifier ID_0 is, by convention, the leader of the pool. Finally, we opt for this replication strategy as it facilitates fast failure recovery. A newly joined pool member is capable of retrieving all metadata from the pool with just one message, whereas the traditional Chord DHT replication usually requires R (replication factor) messages.

$$ID_i = (ID_x + i \cdot \frac{N}{R}) \bmod N, i = 0..R \tag{1}$$

3.2 Migration Algorithm

Data replication in a distributed environment can be impaired by various environment related issues, such as network link failures or congestions, and can result in inconsistent shared service provider metadata in the pool. Consequently, provided with outdated data, SR-nodes might derive false assumptions on the monitored nodes and behave incorrectly in certain situations. For example, provided with outdated metadata on subjected workers, SR-nodes might, in the process of refilling some pool, miss on the fact that some of the newly reassigned workers are currently

Algorithm 2: `migrate(whitebord)`

```
Data: WB, a concrete instance of CompositeServiceState
1  begin
2      nextActivity ← WB.getNextServiceActivity()
3      listOfPeers ← getPeersOfType(nextActivity)
4      nextPeer ← chooseRandomly(listOfPeers)
5      S_ID ← hash(WB_ID)
6      beginPaxos()
7      if nextActivity ≠ JoinActivity then
8          WB.setNextServiceActivityPeer(nextPeer)
9          deliverDirectly(nextPeer_ID, WB)
10     end
11     node ← routeToClosestPrecedingNode(senderID)
12     deliver2Ring(S_ID, WB)
13     commitPaxos()
14 end
```

actively participating to process execution and are thus subject to monitoring. Therefore, for the sake of consistency of any kind of data and thus resulting correct self-healing execution, the replication protocol of Safety Ring is enriched with transactional guarantees. In detail, all replication activities inside the replica pool of the Safety Ring are enclosed within a novel migration algorithm that is based on the Paxos Commit [GL06] transactional protocol. Safety Ring adopts a solution similar to that presented in [SRHS10a], which assumes a symmetric replication scheme in conjunction with a modified version of the paxos commit protocol. A simple example of the general migration algorithm is given in Algorithm 2.

Consequently, the `deliver2Ring(`S_{ID}`, WB)` primitive shown in Algorithm 2 is also enriched with transactional guarantees: whenever a message is delivered to the responsible SR-node, the message is guaranteed to have been received also by all members of the related pool. However, only the directly responsible (i.e., the leader) for the identifier associated with the message handles the received message (e.g., starts monitoring), whereas the other members only store it.

We have identified three cases in which providing consistent data at all replica sites can be impaired most severely by communication disruptions and which have to be ensured by the presented migration algorithm. The first case corresponds to a regular whiteboard migration activity between any two nodes (e.g., step #4 in Figure 1). In this case, a worker can, upon completion of its activity, be certain that, by using the transactionally enforced migration algorithm, the succeeding worker, as well as the SR-pool monitoring it, will acquire a consistent version of the whiteboard and that the execution of the current process will continue reliably.

The second case corresponds to a join migration activity. The join migration activity is a specific case of the regular migration activity since multiple workers have to migrate their results to one and the same node if the process execution is to proceed. In the case of a join node failure, due to late binding, this kind of migration can lead to situations in which actually not all of the workers join on the same node. For example, due to unfortunate join node temporary failure timing, some of the workers might decide to join on a substitution join node, whereas the rest might decide to join on the recovered join node. In both cases, join nodes postpone the execution

of the process until all of the workers have joined on them, and as this is not going to happen, the execution results in a deadlock. Such situations can be successfully overcome with the help of reliable coordinators (possibly they can fail as well), which coordinate the workers on where to reliably migrate their results. Since the SR-nodes are resilient to failures, they can be utilized to perform the coordination task. Therefore, parallel workers can, by using the the migration algorithm, consistently migrate their whiteboards to any –but the same– SR-node, that, once all the whiteboards have been collected and merged, resumes the execution by migrating to the next service provider as in the regular whiteboard migration case.

In the final case, the algorithm is exploited for fault handling. In the event of a worker node failure, the monitoring SR-node can consistently update the node monitoring information of the pool by migrating the whiteboard, obtained from a SM-node (key-value store), to the replacement service provider by means of the migration algorithm, just like in the regular whiteboard migration case. Note, that although there are three different contexts in which it is applied, the algorithm always remains the same.

4 Evaluation

In this section, we present the evaluation of Safety Ring's scalability characteristics in terms of performance and fault tolerance in the context of 40 equally equipped, resource limited devices. For the evaluation, an abstract sample process has been defined that is derived from the firefighter rescue scenario presented in Section 1 and that mimics data back-up in a mobile environment. This back-up process, which is instantiated periodically, transfers data produced on mobile devices to reliable back-up servers. Figure 4 depicts this sample process which consists of six resource undemanding activities (service invocations) of different types and their order of execution (solid arrows). The *Data Cleaning* activity simulates a noise data reduction service. The *Back-Up Coordination* activity represents a service that prepares the actual backup operation. Subsequently, the *Peer Monitor* and *Data Converter/Compressor* activities are invoked in parallel. The Peer Monitor service aims at finding reliable replication sites by simulating a smart peer tracker that additionally keeps statistics on the reliability of the tracked peers while the Data Converter/Compressor service reformats the collected data into a desired format. Finally, the *Replication* activity represents a reliable data back-up service that locally persists the given data. By simulating these resource undemanding services residing on resource limited devices, each service invocation, independently of the service type, is supposed to last for one second, which makes the net execution time add up to five seconds in total (parallel invocations are counted as one). To achieve a constant migration of whiteboards between hosts (and also to take into account the limitations in terms of the hardware resources considered in our example), we only allow the deployment of one service instance at each host at a time. As a consequence, no host is capable of performing more than one activity locally, and has to forward its activity results (whiteboard) to another node determined by late binding in the course of the execution of each process instance.

To host the service instances of the sample process, we consider a node environment that consists of a large number of equally equipped nodes, in terms of main memory, CPU, and physical storage. We have found such an environment in the Amazon Elastic Compute Cloud (Amazon EC2). The unreliability of the mobile environment will be simulated by random failures that are injected into this environment. Within EC2, all nodes are virtual image instances that comprise the same amount of hardware and software resources. However, being only virtual image instances,

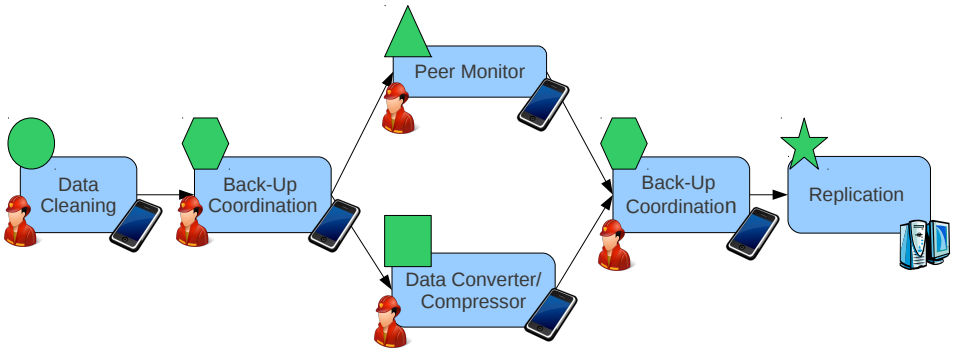

Figure 4: Example Back-Up Process

nodes also exhibit similar message delivery times as their virtual image instances usually reside on one powerful physical machine. The important topic of unequal message delivery times is outside of the topic of this technical report and is along to the evaluation on real physical mobile devices an important subject of future research. A detailed overview the EC2 instance configuration employed in our evaluation is given below:

- Instance type: Medium Instance 32 Bit (c1.medium)
- Operating system: Ubuntu 10.04 LTS
- Main memory: 2.0 GB
- CPU: 5 EC2 Compute Unit (2 virtual cores with 2.5 EC2 Compute Units each)
- Storage: 160 GB
- I/O Performance: Moderate

We have chosen this instance configuration as it nearly corresponds, in terms of CPU power and main memory capacity, to the configurations of the latest off-the-shelf mobile devices (smartphones) [Gao].

4.1 Evaluation Parameters

In order to systematically measure the performance and robustness impact of the Safety Ring, we consider the following configuration parameters as part of our evaluation environment:

- N — This parameter corresponds to the overall number of nodes in the system. These nodes host service instances and thus participate to the distributed execution. We vary N in the range between 10 to 40. In subsequent experiments, the increment of N is set to 10.
- S — This parameter corresponds to the number of all SR-nodes in the system, i.e., nodes that participate to the monitoring of worker nodes during the course of a distributed process execution. The value of S varies between 50% and 100% of N and is incremented in subsequent evaluation runs by 10%.

- I — This parameter corresponds to the number of all concurrently running process instances in the system. Precisely, it represents the number of nodes out of N that start their execution of the process instance at the same time. At such a node, a process instance is only started after the previous invocation that originated from the same node has terminated. For each instance, the resulting execution time is recorded. Hence, the value of I is variable in relative terms within the range of 50% - 100% of N. In subsequent evaluation runs, we increment the value of I by 10%.

- F — This parameter corresponds to the number of failed services during the course of an execution of a process instance. Its value is variable in absolute terms within the range of 0 (no failure in any process instance) to 6 (each service invocation inside a process fails). The value by which this parameter is increased is 1.

Safety Ring attempts to jointly provide a high degree of performance and robustness. However, these goals are not independent of each other but are rather negatively correlated. For instance, it is expected that the additional effort of the SR-nodes needed to reliably store the whiteboard and to perform other fault tolerance tasks for each activity of a process execution will influence the overall performance negatively. The performance of a process could be further affected due to insufficient resource assignment to it by the responsible SR-node caused by the handling of other concurrent instances. However, since we only introduce one instance at a time per node the increase of load on the SR-nodes is expected to be rather moderate and should be easily compensable by the increase of S. In general, it is expected that the increase of I will worsen the execution times (higher load), whereas the increase of S is supposed to improve the execution times (more handlers to distribute the load). Also, the increase of N should lead to improved execution times as consequently more service providers are added to the system relieving thereby the already existing ones. In the event of node failures, execution times are expected to be higher due to the necessary recovery effort. In such cases, however, the resulting bad performance should be only reflected on the processes that the failed nodes were servicing, whereas for the unaffected processes the performance should stay the same. To investigate the performance and robustness of the system and the correlation between the two aspects, we will address in the evaluation the following open questions:

- What is the upper bound value of each evaluation parameter S, N, I, F for which the execution times start to deteriorate with the given resources?

- What is the optimal ratio of S relative to N in terms of minimal execution times?

- What is the optimal ratio of S relative to F in terms of maximal numbers of F?

The initial values of the configuration parameters are set to $N = 10$, $S = 5$, $I = 5$, and $F = 0$. We proceed by increasing the values of only one single parameter at a time and by measuring the resulting execution times of the running processes in the system. Once we have evaluated all the possible parameter value combinations we investigate the correlation between them with regard to the stated open questions. We start our evaluation process with N, and then address S, I, and finally F.

4.2 Evaluation Results

Figures 5, 6, 7, and 8 illustrate the evaluation results of four failure-free runs of our sample process. Each run is characterized by the the total number of nodes N to be evaluated (with

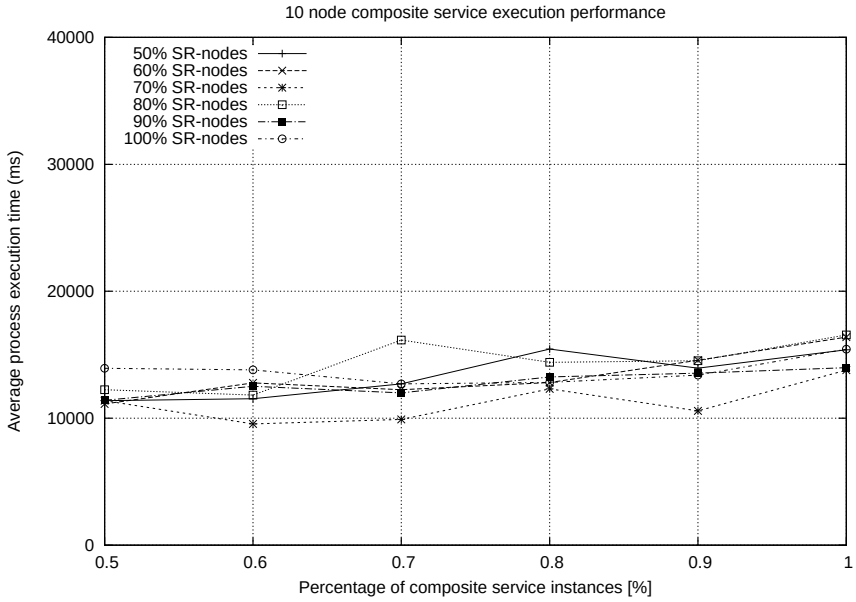

Figure 5: 10 Node Distributed Process Execution Performance

$N \in \{10, 20, 30, 40\}$) which stays constant throughout the entire run. Furthermore, each run is additionally divided into several cases which are defined by the percentage (50%, 60%, 70%, 80%, 90%, 100%) of the nodes that act as SR-nodes (S) at the same time, and by the percentage of (50%, 60%, 70%, 80%, 90%, 100%) of the nodes (I) that concurrently start their instance of the example process. For each case graph, S stays constant, whereas I increases.

We can observe that already the 10 node run (Figure 5) exhibits the expected moderate performance decrease behavior. That is, with the increase of I, the average execution times increase only slightly for all S cases. Moreover, all S cases show similar execution times. In the 20 node run (see Figure 6), the average execution times for most cases continue to increase linearly at a small rate ($\sim$ 10% on avg.), but for some cases (e.g., 50%, 90%) the increase slightly oscillates, which can be explained by the fact that the doubled load on the system –in terms of I– is unevenly distributed among all SR-nodes as the selection of SR-nodes is determined by the SR-nodes' assigned area of responsibility inside the ring topology. For instance, in our experiments, we applied a 24-bit (i.e., $2^{24} \sim 16.8$ million) integer address space for the mapping of nodes. In such a vast address space, induced by the mapping of the SHA1 hash function, frequently nodes could be found whose responsibility area, when compared against the whole address space, was approximately less than 1% (e.g., the responsibility for a 100000 integer address space out of the entire 16.8 million address space). Clearly, by possessing such low responsibility areas the corresponding SR-nodes are less likely to be addressed for monitoring tasks than SR-nodes with significantly larger responsibility areas. Therefore, unequal responsibility areas in the ring directly result in uneven load among all SR-nodes. In the 30 node run (see Figure 7), the execution times continue to linearly increase at approximately the same rate as in in the 20 node run. However, the oscil-

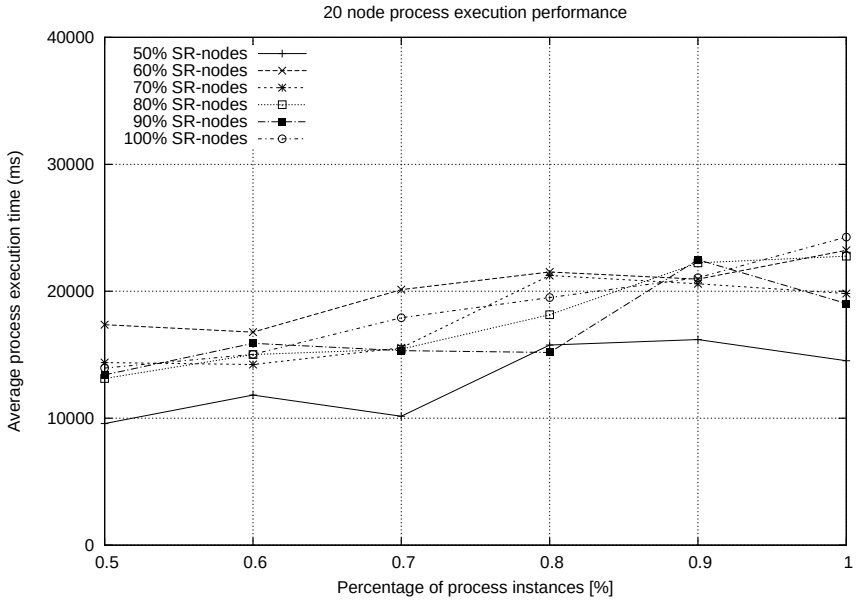

Figure 6: 20 Node Distributed Process Execution Performance

lations in this run become greater which can be answered with the same explanation as in the 20 node case. Again, the increased load will be unevenly distributed among the SR-nodes causing always different execution times for each instance. Apparently, in those cases the introduction of new SR-nodes into the system did not significantly contribute to the redistribution of the load. In the 40 node run (see Figure 8), execution times are again slightly higher than in the 30 node run, but the increase rate is also slightly lower than in the previous case. However, it shows that for the lower SR-node percentage cases (e.g., 50%, 60%) high load results in bad performances as they provide less SR-nodes for the handling of high process instance numbers.

Figure 9 illustrates the evaluation results of the sample process runs in the event of node failures. The runs are divided into several cases, which are defined by the number of evaluated nodes N (i.e., 10, 20, 30, 40) and the number of failed service hosts (F). Moreover, for each case S and I are defined as well, however they stay constant and the maximal value is chosen (each being 100%), whereas only F increases. From the graph, we can observe that the average execution times heavily differ and oscillate in value with the increasing number of node failures (F). This can be explained by unequal fault detector heartbeat timing and the fact that some of the failed SR-nodes had more workers assigned to them than the others, which implies uneven SR-node fault handling effort (e.g., replica pool recovery) for all SR-nodes. In some cases (e.g., 10 nodes) the system was not capable of recovering during the execution from 6 consecutive node failures as all service instances that would have been able to continue the execution were not available.

Based on evaluation results learned from the graphs, and with respect to the stated configuration questions, we conclude that the system expectedly shows for all cases linearly degrading performance ($\sim$ 10% on avg.) with the increase of I in a failure-free scenario (and in the presence

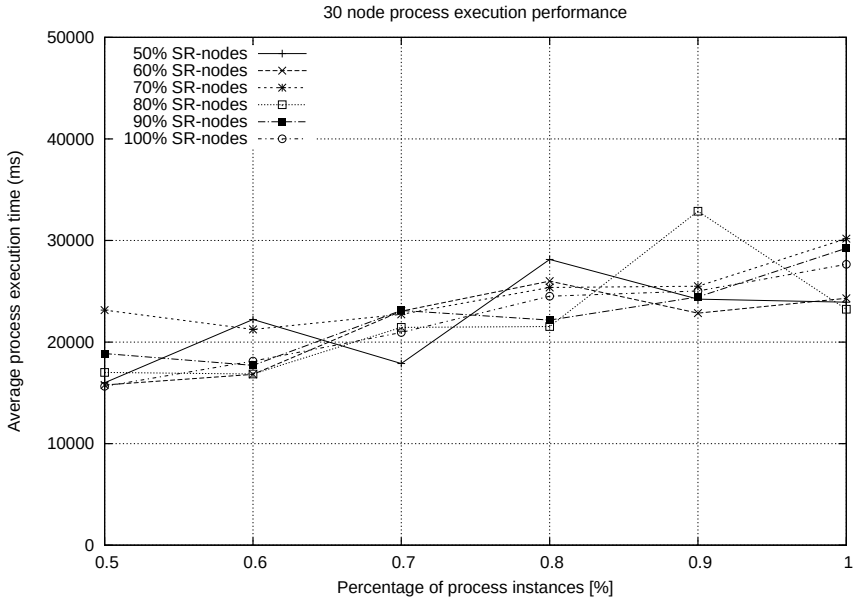

Figure 7: 30 Node Distributed Process Execution Performance

of redundancy). However, the evaluated performance also shows slight oscillations and inclines of execution times which is in our opinion a direct result of the absence of a load balancer in our experiments and suboptimal SR-node selection and distribution in the ring topology. As it turns out, the pub/sub repository driven load balancing used for late-binding of workers is incompatible with the SR-node ring structuring and thus requires a separate solution, which is a subject of future research. Hence, SR-nodes are assigned process instances for handling according to their unevenly distributed responsibility area in the ring topology. Clearly, SR-nodes that have small ring responsibility areas do not contribute very much to the execution of processes, and their existence in the ring topology is questionable as they rather inflict additional overhead on the system (e.g., ring maintenance, ring construction etc.). Therefore, an optimal distribution of the running process instances among the SR-nodes, independent of the ring area responsibility and driven by a load balancer, should result in less oscillating and generally improved execution times. In the absence of a load balancer, and based on the observations from the graphs, we suggest to keep S at a high value in order to accommodate for the increase of I in terms of performance and robustness, however with regard to an approximately even ring responsibility area distribution among the SR-nodes.

5 Related Work

This section reviews related work addressing the reliable distributed service execution, and scalable and reliable data stores for the management of large volumes of data.

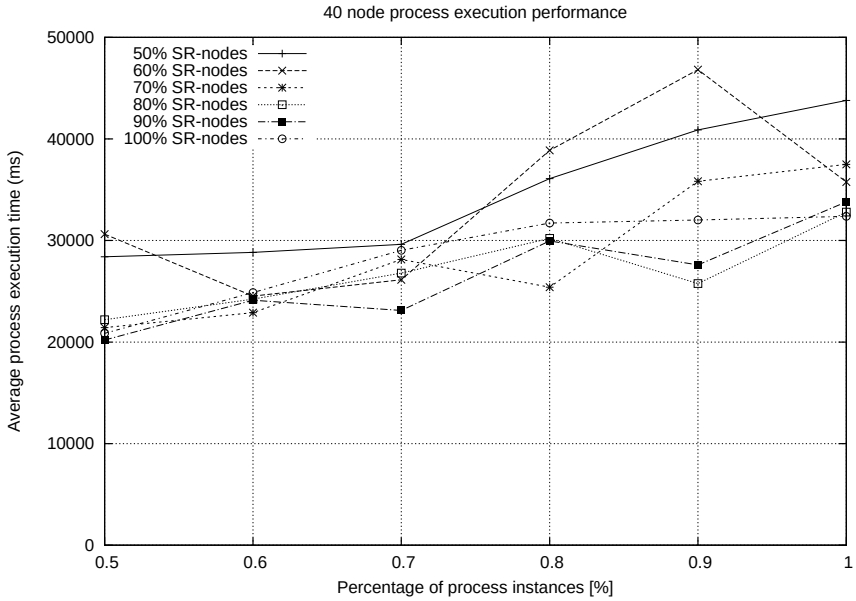

Figure 8: 40 Node Distributed Process Execution Performance

Providing reliable distributed process execution presents a challenging task, and there are many solutions to it. In general, most of the approaches are based on either replication techniques or rollback recovery techniques. In replication-based approaches, either recovery data or execution code are placed at various replacement node sites and exploited during recovery. This approach is characterized by issues such as consistency of replicas, expensive replacement peer agreement algorithms, and economic resource utilization [Ye06, Yu10]. In rollback recovery approaches [KCS10, JKG09], execution related data is backed-up at remote stable storage and recovered in the event of failure. The challenge with this approach is to find an optimal backup frequency which does not degrade the execution performance but guarantees reasonable recovery times. OSIRIS' Safety Ring offers both approaches. The provided key-value store can be used as a scalable stable storage for data backups, whereas the replication performed on the Safety Ring is guaranteed to be consistent and effective in terms of peer agreement. In terms of fault handling, most existing approaches are, similar to OSIRIS-SR, based on dedicated nodes for the sake of monitoring and recovery [Yu10, FJLM05]. Unlike our approach where any node can take the role of the dedicated fault handling node (SR-node), in [Yu10, FJLM05] fault handling nodes are predefined and their sensibility to failures is not considered. Moreover, approaches like [AUS+09] that allow for all nodes to fail are based on complex node consensus algorithms for the election of replacement nodes, whereas in Safety Ring, induced by the ring topology, leader election is simple and scalable.

Although there are many approaches that offer scalable storage systems [LM10, DHJ+07, BBC+11] for large volumes of data, they generally opt for sacrificing consistency of the data they manage for the sake of availability. In the key-value store of Safety Ring the focus lies rather on

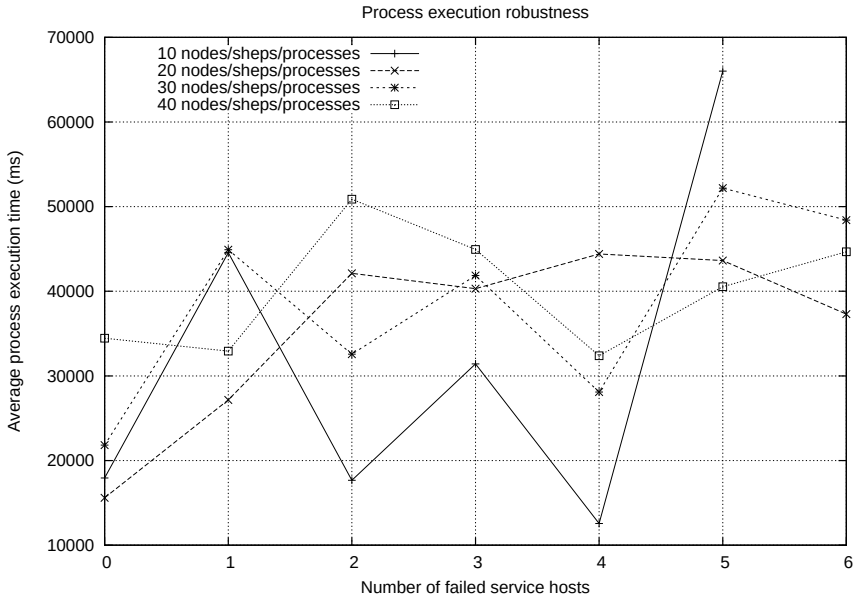

Figure 9: Distributed Process Execution Robustness

consistency as in [RST11], because only consistent data at replacement nodes ensures a correct failure handling. Similar to [RST11], we leverage Paxos for database replication. However, we apply a concrete Paxos protocol [SRHS10b] that is based on a symmetric replication scheme which allows us for a faster fault handling in the event of a failure.

6 Conclusion

The execution of processes which are built by (recursively) composing existing distributed services has to be provided in a way that jointly guarantees fault-tolerance and a high degree of performance and scalability. In terms of scalability, the middleware orchestrating process instances needs to be inherently distributed, to a void a single point of failure and a performance bottleneck. Hence, when deploying services and/or the middleware for executing processes, failure handling needs to take place in a self-healing way, without manual intervention. This is an important demand in all types of IT environments, and particularly in cases where unreliable, resource-limited (and mobile) devices are involved.

In this technical report, we have presented the OSIRIS Safety Ring, a middleware that addresses the aforementioned issues of distributed service support. Scalable orchestration and execution of processes is achieved in Safety Ring by following a purely peer-to-peer based continuation passing style approach. In particular, every peer based only on locally available knowledge (i.e., without having to rely on centralized components), contributes to process execution. Moreover, metadata is managed by means of a reliable and scalable key-value data store. In order to be able

to support self-healing distributed execution we have identified critical service host failure scenarios and presented a solution to them in the form of a Safety Ring, a sophisticated fault handling mechanism. The presented Safety Ring is based on a self-organizing node overlay composed of replaceable dedicated monitoring nodes, that attend to active service instance hosts. Reliability is also provided by a Paxos protocol which guarantees the availability of replicated instance metadata of a process execution. Finally, in an evaluation based on amazon EC2 resources, we have shown that the high level of robustness and the self-healing behavior has acceptable effects on Safety Ring's scalability characteristics.

In our future work, we plan to expand the Safety Ring approach to streaming services, i.e., to applications that continuously produce and process data. Essentially, these applications have strong demands for a high degree of reliability and are likely to be deployed on resource-limited (mobile) devices, e.g., in the context of sensor networks. In addition, we aim at further improving the self-healing features of OSIRIS' Safety Ring by introducing a load balancer which would allow for an even load distribution among the SR-nodes. Finally, the Safety Ring is to be enabled with run-time service instance instantiation and deployment capabilities driven by a comprehensive economic model, jointly taking into account the cost for service provisioning and system parameters such as the load of service providers or network bandwidth.

7 Acknowledgment

This work has been partly funded by the Swiss National Science Foundation (SNF) in the context of the *SOSOA* Sinergia project (Self-Organizing Service-Oriented Architectures) under contract no. CRSI22_127386/1.

References

[AUS+09] K. Abe, T. Ueda, M. Shikano, H. Ishibashi, and T. Matsuura. Toward Fault-Tolerant P2P Systems: Constructing a Stable Virtual Peer from Multiple Unstable Peers. In *Proceedings of the 1st International Conference on Advances in P2P Systems (AP2PS'09)*, pages 104 –110, October 2009.

[BBC+11] Jason Baker, Chris Bondç, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In Gerhard Weikum, Joseph Hellerstein, and Michael Stonebraker, editors, *Proceedings of the 2011 Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, January 2011.

[BS11] Gert Brettlecker and Heiko Schuldt. Reliable Distributed Data Stream Management in Mobile Environments. *Information Systems Journal*, 36(3):618–643, 2011.

[CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011.

[CWEF06] Liang Chen, Bruno Wassermann, Wolfgang Emmerich, and Howard Foster. Web Service Orchestration with BPEL. In *Proceedings of the 28th International Confer-*

ence on Software Engineering, ICSE '06, pages 1071–1072, Shanghai, China, 2006. ACM.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's Highly Available Key-Value Store. In *Proceedings of the ACM Symposiumon Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[DK76] F. DeRemer and H.H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[FJLM05] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES Distributed Publish/Subscribe System. In *Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems*, pages 12–30, 2005.

[GAH07] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *Proceedings of the 2005/2006 International Conference on Databases, Information Systems, and Peer-to-Peer Computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[Gao] Shengkui Gao. A Survey of Latest Performance, Development and Measurement Issues of Smart Phones Design, http://www.cse.wustl.edu/ jain/cse567-11/ftp/smartphn/index.html.

[GL06] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006.

[JKG09] S. Jafar, A. Krings, and T. Gautier. Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing. *IEEE Transactions on Dependable and Secure Computing*, 6(1):32 –44, January-March 2009.

[KBG+10] Martin Keen, Bryan Brown, Andy Garratt, Benjamin Käckenmeister, Ahmed Khairy, Kevin OMahony, and Lei Yu. *Building IBM Business Process Management Solutions Using WebSphere V7 and Business Space*. Number SG24-7861-00 in IBM Redbooks. IBM, May 2010.

[KCS10] R. Kaur, R.K. Challa, and R. Singh. Antecedence Graph based Checkpointing and Recovery for Mobile Agents. In *Proceedings of the IEEE International Conference on Communication Control and Computing Technologies (ICCCCT'2010)*, pages 419 –424, October 2010.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a Decentralized Structured Storage System. *SIGOPS Operating System Reviews*, 44:35–40, April 2010.

[MR08] Boris Mejías and Peter Van Roy. The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay Networks. *Parallel Processing Letters*, 18(3):411–432, 2008.

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16:389–415, July 2007.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[RST11] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a Scalable, Consistent, and Highly Available Datastore. *Proceedings of the VLDB Endowment*, 4:243–254, January 2011.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 149–160, New York, NY, USA, 2001. ACM.

[SRHS10a] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pages 448–454. IEEE Computer Society, 2010.

[SRHS10b] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced Paxos Commit for Transactions on DHTs. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 448–454, Washington, DC, USA, 2010. IEEE Computer Society.

[SST+05] C. Schuler, H. Schuldt, C. Türker, R. Weber, and H.-J. Schek. Peer-to-peer Execution of (transactional) Processes. *International Journal of Cooperative Information Systems*, 14(4):377–406, 2005.

[STS+06] C. Schuler, C. Türker, H.-J. Schek, R. Weber, and H. Schuldt. Scalable Peer-to-Peer Process Management. *International Journal of Business Process Integration and Management (IJBPIM)*, 1(2):129–142, 2006.

[Ye06] Xinfeng Ye. Towards a Reliable Distributed Web Service Execution Engine. In *Proceesings of the International Conference on Web Services (ICWS '06)*, pages 595 –602, September 2006.

[Yu10] Weihai Yu. Fault Handling and Recovery in Decentralized Services Orchestration. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 98–105, New York, NY, USA, 2010. ACM.