OSIRIS-SR: A Safety Ring for Self-Healing Distributed Composite Service Execution

Nenad Stojnić Heiko Schuldt
Department of Mathematics and Computer Science
University of Basel, Switzerland
{*nenad.stojnic*|*heiko.schuldt*}@*unibas.ch*

***Abstract*—The advent of service-oriented architectures has strongly facilitated the development and deployment of large-scale distributed applications. The middleware for orchestrating 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 any single point of failure. Self-healing execution of composite services requires replicated control metadata and instance data in a way that does not affect adaptivity and elasticity of the middleware. In this paper, we present OSIRIS-SR, a decentralized approach to self-healing composite service execution in a distributed environment. OSIRIS-SR exploits dedicated node monitors, organized in a self-organizing Safety Ring, for the replication of control data. Moreover, OSIRIS-SR leverages virtual stable storage for managing composite service instance data in a robust way. We present the architecture of OSIRIS-SR's Safety Ring and discuss how it provides self-healing composite service execution. The performance evaluation shows that the additional gain in robustness has only marginal effects on the scalability characteristics of the system.**

I. Introduction

The success of service-oriented architectures has paved the way for an important class of complex distributed applications that are built by combining existing services. As these composite services, or workflow processes, are used more and more in domains where their reliable execution is crucial, they need to run in a robust and highly scalable way. This applies both to the composite services and the middleware which orchestrates their execution. Hence, while composite services are inherently distributed, the middleware supporting their execution also has to be distributed, to avoid any single point of failure or potential performance bottleneck. However, such a distributed middleware poses new challenges in terms of robustness, reliability and advanced deployment (e.g., mobile, resource limited environments). This is especially true compared to traditional centralized middleware solutions [1].

In this paper, we introduce *OSIRIS-SR*, a middleware for the decentralized and distributed management of composite services based on locally available metadata (e.g., available service hosts, instance data of composite services, etc.). OSIRIS-SR has been built to combine both reliability and robustness, needed for business-critical applications [2] with a small systems footprint, so that it can even be used with unreliable commodity hardware [3] (e.g., mobile devices). The continuation model of distributed execution leveraged in OSIRIS-SR requires redundancy to prevent the loss of execution-related metadata, and thus unnecessary repetition of possibly expensive computation. As the volume of the replication metadata increases with the number of concurrently running composite service instances, the resources available need to be used efficiently, and the system should be able to scale with the number of nodes and concurrent composite service instances. In this paper, we present a scalable approach to self-healing data management within composite service instances The solution adopted is based on the concept of a *Safety Ring*, a scalable self-organizing "node monitor" overlay 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.

The remainder is organized as follows: Section II introduces the basics of OSIRIS-SR. Section III presents the Safety Ring for self-healing decentralized composite service execution which is subject to evaluations in Section IV. Section V presents related work and Section VI concludes.

II. System Model

OSIRIS-SR is an extension of OSIRIS [2] in which every node can take over one or several roles: i.) provider of a service; ii.) host ("worker") for the OSIRIS-SR middleware for decentralized execution and orchestration of distributed composite services; iii.) dedicated node fault handler, in which case it is called SR-node; and iv.) data management node, in which case it is referred to as the SM-node. By default, any OSIRIS-SR node is at start-up equipped with the data management role and the composite service support role, whereas the additional fault-handling role is determined randomly based on the number of the already existing fault-handling nodes.

From an architectural point of view, all the OSIRIS-SR roles and their respective functionalities are implemented within three software layers (c.f. Figure 1) that can be deployed in any combination, at any node in the network. The first layer, the core composite service support layer (Figure 1.a), corresponds to the decentralized execution and orchestration role and provides local functionality for composite service management in terms of invocation, navigation, and routing of service instances. The second layer, the OSIRIS-SR layer (Figure 1.c), corresponds to

the fault handler role and provides local functionality for node monitoring, data consistency enforcement, ring topology construction, and failure recovery. The third layer, the Shared Memory layer (Figure 1.b), corresponds to the data management role and provides node functionality for reliable and distributed data dissemination and storage.

Finally, we assume communication channels to be reliable channels. Services crash with a crash-stop behavior, an eventually perfect failure detector [4] is available at any node and all services invoked by worker nodes are fail-safe.

A. Distributed Composite Service Execution

An OSIRIS-SR composite service represents an description of an ordered set of activities, each of them corresponding to the invocation of a 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 instances is decided entirely by the local OSIRIS-SR layer at run-time (late binding), depending on the current configuration of the system (availability, load, cost of invocation). Following the paradigm of service-oriented architectures [5], composite services 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 composite service instance, called the *whiteboard*, to the service request parameters and back.

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 composite service definition. Upon completion of a service at a node, the control over the execution migrates to one or more successor nodes, by delivering a migration token containing flow-control information and the whiteboard. In order to participate to distributed composite service execution, nodes deploy the service instance support software layer as depicted in Figure 1.a. Further details, illustrated on an example composite service, as on how the continuation passing distributed execution works can be found in [6].

The decentralized orchestration of composite services is supported by locally available system-related metadata at all involved nodes (SM-nodes). This includes metadata used for routing (e.g., node addresses, hosted services at nodes), load balancing (workload of other nodes) and activity result backups (whiteboards). The storage and management of all necessary metadata relies on two mechanisms. One is a publish-subscribe repository and the other is a key-value store. The publish-subscribe repository is responsible for the pre-execution timed collection and dissemination of metadata at nodes [2]. The key-value store serves as permanent and reliable stable storage for any kind of data, but in particular to the whiteboard.

B. Self-Healing Distributed Execution

The Safety Ring (see Figure 1.c) mechanism, employed for scalable and reliable fault-handling, is based on the idea

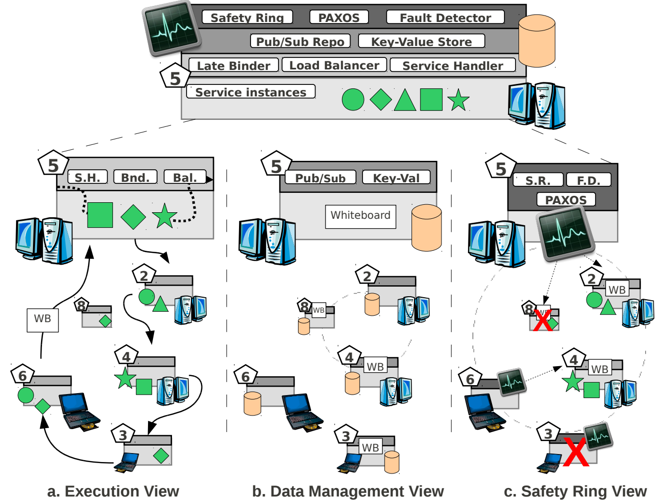

Figure 1. OSIRIS-SR: Architectural Layers

of making each active worker node subject to supervision by one SR-node. The responsible SR-node, in turn, is in charge of worker node monitoring and failure recovery. In the event of a node failure, the designated responsible SR-node elects a replacement service host of the same type (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 key-value store) as the crashed node, and the failed service is restarted. Finally, any intermediate results produced on the whiteboard by the crashed node is discarded from the key-value store.

To guarantee continuous 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, and is substituted by it.

Scalable and unanimous 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 (unique and ordered) subset of selected nodes, which allows for overall effective node interaction.

The consistency of all data (whiteboard, execution control data, etc.) shared by multiple nodes (e.g., SM-nodes) is provided in OSIRIS-SR with a novel migration algorithm, that is enforced with transactional guarantees. The new migration algorithm replicates the whiteboard along to the worker, responsible of continuing the execution, also to the replica pool (SR-nodes), which informs the pool members that a new worker is subject to monitoring. Further details on how the self-healing distributed execution works can be found in [6].

Finally, the introduced self-healing execution model al-

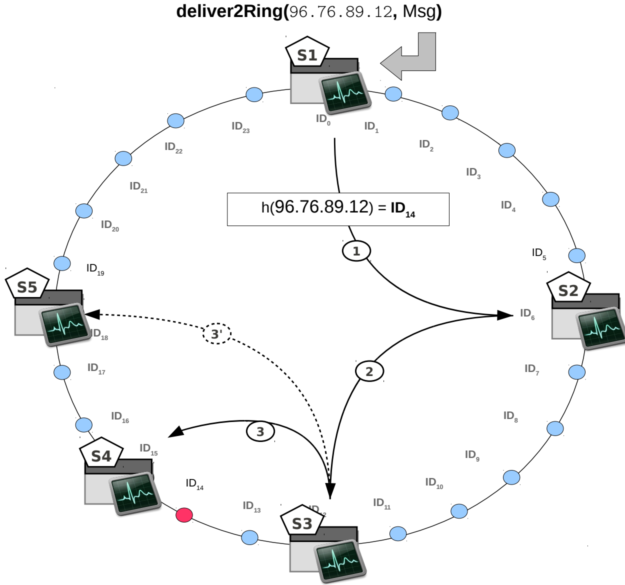

Figure 2. Node Ring Topology

ways assumes an adequate number of replacement service hosts and SR-nodes for fault recovery. The problems of having insufficient service providers or SR-nodes at disposal are outside of the scope of this paper. However, this can be overcome with dynamic service deployment techniques in the former case and by empowering the SR-nodes to promote regular workers to SR-nodes in the latter case.

III. Safety Ring

OSIRIS-SR aims at a large spectrum of possible deployments – from a small number of powerful servers up to a 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 in dynamic and large environments have to be provided. Even though each active worker node is constantly monitored by one SR-node, additional substitution SR-nodes have to be provided 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 result in an effective SR-node reassignment. Such properties can effectively and efficiently be provided by leveraging DHT-based node ring topologies such as Chord [7].

Given the unique circular space mapping, imposed by the ring topology, any worker node will always lie in the circular identifier space between two SR-nodes. Hence, the Safety Ring performs worker to SR-node assignment by subjecting all workers located between two SR-nodes to the SR-node with the higher circular ID. Figure 2 illustrates a sample Safety Ring and the positioning of nodes in the ring space, in which case the smaller circles correspond to worker nodes, the bigger rectangles to SR-nodes, and the ID node labels to their respective circular identifier. For example, the worker with the circular ID_{14} is located between SR-nodes with the identifiers ID_{15} and ID_{12}, with the SR-node ID_{15} being its supervisor as it possesses a higher circular identifier than the other SR-node. Since workers are only temporarily associated with SR-nodes (i.e., during a service activity), their participation to the ring construction would result in high ring churn rates. Hence, the Safety Ring is built out of SR-nodes only. More details on Chord ring construction (maintenance) can be found in [7]. Being only logically a member of the ring topology, worker nodes can nevertheless interact with their SR-node by means of the `deliver2Ring(IP, Message)` primitive. `IP` 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 among SR-nodes until it reaches the responsible SR-node for the worker identifier associated with the message. Algorithm 1 illustrates the functionality of the `deliver2Ring(IP, Message)` primitive. The communication in the ring relies on the efficient Chord routing algorithm. Moreover, Figure 2 depicts the forwarding of messages (solid arrows 1, 2 and 3) induced by the functionality of `deliver2Ring(IP, Message)`. 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 is reassigned accordingly. Moreover, for performance reasons, the Safety Ring features the primitive `deliverDirectly(IP, Message)` as well, which allows workers to communicate messages to destination nodes directly (now, `IP` corresponds to the identifier of the destination node), without having to route the messages in the Safety Ring.

Note that resorting to a ring structure comes with the price of uneven ring chunk distribution, churn, Byzantine nodes etc. However, those problems are outside of the scope of this paper, and can be overcome by applying solutions as presented in [8].

A. Replica Pool

For the purpose of recovering failed workers each SR-node maintains a certain amount of service execution related metadata on its subjected worker nodes. This metadata includes information such as the current activity of the worker and the activity input whiteboard. Replication of this metadata in the Safety Ring prevents the loss of it in the event of the responsible SR-node failure. The set of the metadata sharing SR-nodes we name the *Replica Pool*. Within the pool we designate a *leader* that is in charge of updating (managing) the shared metadata, whereas the role of the other nodes is to serve as data back up hosts. When a pool member fails, the succeeding SR-node in the Safety Ring detects this failure and substitutes the failed SR-node by taking over its position in the pool and retrieving the shared metadata from the rest of the pool. In order to dynamically determine the pool members we resort

Require: IP identifier of the sender and Message to be delivered
$senderID = hash(IP)$
$ownID = getOwnID()$
$predID = getPredecessorID()$
if $predID < senderID$ **and** $senderID \leq ownID$ **then**
$handle(Message)$
else
$node = routeToClosestPrecedingNode(senderID)$
$node.deliver2Ring(IP, Message)$
end if

Algorithm 1. `deliver2Ring(IP, Message)`

Require: WB, a concrete instance of CompositeServiceState
$nextActivity = WB.getNextServiceActivity()$
$listOfPeers = getPeersOfType(nextActivity)$
$nextPeer = chooseRandomly(listOfPeers)$
$S_{ID} = hash(WB_{ID})$
$beginPaxos()$
if $nextActivity \neq JoinActivity$ **then**
$WB.setNextServiceActivityPeer(nextPeer)$
$deliverDirect(nextPeer_{ID}, WB)$
end if
$node = routeToClosestPrecedingNode(senderID)$
$deliver2Ring(S_{ID}, WB)$
$commitPaxos()$

Algorithm 2. `migrate(whitebord)`

to *Symmetric Replication* [9]. 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* the desired replication factor, and *i* 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, this replication strategy facilitates fast failure recovery. Precisely, 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.

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

B. Migration Algorithm

Data replication in a distributed environment can be impaired by various environment-related issues (e.g., link failures, congestions), and can result in inconsistent shared worker metadata in the pool. Consequently, provided with outdated data SR-nodes might derive false assumptions on the monitored nodes and behave falsely in certain situations. Therefore, for the sake of consistency of any kind of data and thus resulting correct self-healing execution, the replication protocol of the 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* [10] transactional protocol. The Safety Ring adopts a solution similar to that presented in [11], 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.

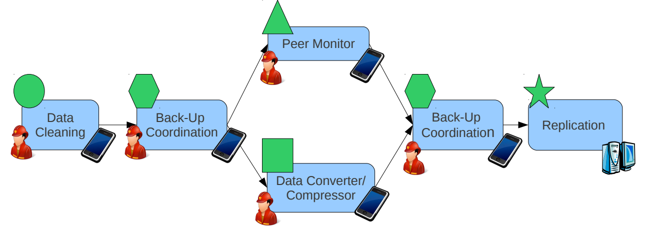

Fig 3. Sample Composite Service

Consequently, the `deliver2Ring(`S_{ID}`, WB)` primitive shown in Algorithm 2 is enriched with transactional guarantees, meaning that 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 pool leader handles the received message (e.g., starts monitoring), whereas the other members only store it. Further details on the migration algorithms, including application contexts, can be found in [6].

IV. Evaluation

The evaluation of OSIRIS-SR focuses on the scalability characteristics of the Safety Ring. For this purpose a composite service is considered that mimics data back-up in a mobile environment, depicted in Figure 3. The invocation of each composition service, independently of the its type, lasts for about one second, which makes the net execution time add up to five seconds in total (parallel invocations are counted as one). Taking into account the limitations, in terms of the hardware resources, of a mobile device (in our considered environment) we deploy only one service instance per host. Further details on the considered composite service can be found in [6]. To host the service instances, we consider an environment of 40 equally equipped (both hardware and software resources) nodes in the Amazon Elastic Compute Cloud (Amazon EC2). In particular, we have chosen the *c1.medium* [12] virtual image 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 (smart phones).

In order to systematically measure the performance and robustness impact of the Safety Ring, we consider the

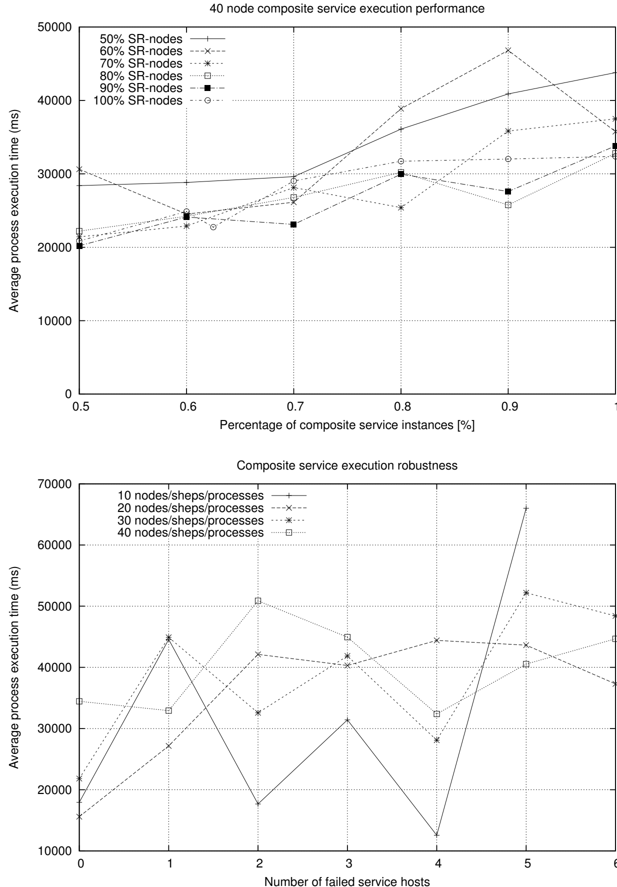

Fig 4. Composite Service Execution Performance

following configuration parameters of our evaluation environment: N (number of nodes), S (number of SR-nodes), I (number of concurrent composite service instances), F (number of failed services). We conduct the evaluation by increasing the values of only one single parameter at a time and by measuring the resulting execution times of the running composite services in the system.

Figure 4 illustrates the evaluation results of 40 node failure free run (upper graph) and a failure prone run (lower graph) of the considered composite service. The 40 node run is divided into several cases which are defined by the percentage (from 50% to 100%) of the nodes that act as SR-nodes (S) at the same time, and by the percentage of (between 50% and 100%) of the nodes (I) that concurrently start their instance of the example composite service. For each case, S stays constant, whereas I increases. We can observe that with the increase of the concurrent composite service instances, the average execution times slightly oscillatingly increase at a small linear rate. Except for some cases (50%, 60%), all the SR-node percentages show similar execution times. The oscillations in increase can be explained by the fact that the additional load, in terms of composite service instances, is unevenly distributed among the nodes as we select both workers and SR-nodes to continue the execution randomly (i.e., we do not use OSIRIS-SR's load balancer). Moreover, it shows that for the lower SR-node percentage cases (50%, 60%) high load results in higher execution times as less SR-nodes are provided for the handling of high composite service instance numbers.

The failure-prone runs are divided into several cases, which are defined by the number of evaluated nodes (N) and the number of failed service hosts (F). Additionally, for each case S and I are defined as well, however they maintain a constant and maximal (100%) value, whereas only F increases. From the graph, we can observe that the average execution times heavily differ with increasing node failure numbers (F). This can be explained by unequal fault detector heart beat timing and uneven (ring based) worker to SR-node assignment, which implies uneven SR-node fault handling effort (e.g., replica pool recovery) for all SR-nodes. In some cases (10 nodes) the system was not capable of recovering from 6 consecutive node failures as all required service hosts, needed to continue the execution, went down. Further and more detailed evaluations can be found in [6].

Based on the observations of the graphs we conclude that the system shows a reasonable scaling behavior (in the presence of redundancy) in a failure-free scenario, however with slight oscillations and inclines of execution times due to the lack of a load balancer (in our experiments) and suboptimal SR-node selection and distribution. In general, the percentage of SR-nodes S in the system should always be kept high, i.e., equal to N both for performance and robustness reasons.

V. Related Work

Providing reliable distributed composite service execution presents a challenging task, and there are many solutions to it. In general, most of the approaches are based on either replication techniques [13], [14] or rollback-recovery [15], [16] techniques. OSIRIS-SR offers both approaches. The provided key-value store can be used as a scalable stable storage for data back-ups, whereas the replication performed on our Safety Ring is guaranteed to be consistent and effective in terms of the 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 [14], [17]. Unlike our approach where any node can take the role of the dedicated fault handling node (SR-node), in [14], [17] fault handling nodes are predefined and their sensibility to failures is not considered. Moreover, approaches like [18] that allow for all nodes to fail are based on complex node consensus algorithms for the election of replacement nodes, whereas in OSIRIS-SR, induced by the ring topology, leader election is simple and scalable.

Although there are many approaches that offer scalable storage systems [19]–[21] for large volumes of data, they generally sacrifice consistency of the data they manage for the sake of availability. In the key-value store of OSIRIS-SR the focus lies rather on consistency as in [22], because only

consistent data at replacement nodes ensures a correct failure handling. Similar to [22], we leverage Paxos for database replication. However, we apply a concrete Paxos protocol [11] that is based on a symmetric replication scheme which allows us for a faster fault handling in the event of a failure.

VI. Conclusion

In this paper, we have presented OSIRIS-SR, a middleware that addresses the scalability and fault-tolerance issues of distributed service support. Scalable orchestration and execution of composite services is achieved in OSIRIS-SR by following a continuation model approach. In particular, every peer based only on locally available knowledge (i.e., without having to rely on centralized components), contributes to composite service 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 presented the 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 composite service 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 OSIRIS-SR's scalability characteristics.

In our future work, we plan to expand the OSIRIS-SR 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-SR by enabling it to dynamically deploy service instances 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.

Acknowledgement

This work has been partly funded by the Swiss National Science Foundation in the context of the SOSOA project.

References

[1] L. Chen, B. Wassermann, W. Emmerich, and H. Foster, "Web Service Orchestration with BPEL," in *Proc. ICSE*, 2006.

[2] C. Schuler, C. Türker, H.-J. Schek, R. Weber, and H. Schuldt, "Scalable Peer-to-Peer Process Management," *IJBPIM*, vol. 1, no. 2, 2006.

[3] G. Brettlecker and H. Schuldt, "Reliable Distributed Data Stream Management in Mobile Environments," *Inf. Syst.*, vol. 36, no. 3, pp. 618–643, 2011.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[5] M. P. Papazoglou and W.-J. Heuvel, "Service oriented Architectures: Approaches, Technologies and Research Issues," *The VLDB Journal*, vol. 16, Jul. 2007.

[6] N. Stojnić and H. Schuldt, "Safety Ring: Fault-tolerant Distributed Process Execution in OSIRIS," University of Basel, Switzerland, Technical Report CS-2012-002, Mar. 2012, available at http://dbis.cs.unibas.ch/publications/.

[7] I. Stoica, R. Morris, D. Karger *et al.*, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proceedings of SIGCOMM*, 2001.

[8] B. Mejias and P. V. Roy, "The Relaxed-Ring: a Fault-Tolerant Topology for Structured Overlay Networks," *Parallel Processing Letters*, vol. 18, no. 3, 2008.

[9] A. Ghodsi, L. O. Alima, and S. Haridi, "Symmetric Replication for Structured Peer-to-Peer Systems," in *DBISP2P*, 2005.

[10] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Trans. Database Syst.*, 2006.

[11] F. Schintke, A. Reinefeld, S. Haridi, and T. Schütt, "Enhanced Paxos Commit for Transactions on DHTs," in *Proc. CCGRID*, 2010.

[12] Amazon AWS, http://aws.amazon.com/ec2/instance-types/.

[13] X. Ye, "Towards a Reliable Distributed Web Service Execution Engine," in *Proc. ICWS*, 2006.

[14] W. Yu, "Fault Handling and Recovery in Decentralized Services Orchestration," in *Proc. iiWAS*, 2010.

[15] R. Kaur, R. Challa, and R. Singh, "Antecedence Graph Based Checkpointing and Recovery for Mobile Agents," in *Proceedings of ICCCCT*, 2010.

[16] S. Jafar, A. Krings, and T. Gautier, "Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing," *IEEE Transactions on Dependable and Secure Computing*, 2009.

[17] E. Fidler, H. a. Jacobsen, G. Li, and S. Mankovski, "The PADRES Distributed Publish/Subscribe System," in *In Proceedings of ICFI*, 2005.

[18] K. Abe, T. Ueda, M. Shikano *et al.*, "Toward Fault-Tolerant P2P Systems: Constructing a Stable Virtual Peer from Multiple Unstable Peers," in *Proceedings of AP2PS*, 2009.

[19] A. Lakshman and P. Malik, "Cassandra: a Decentralized Structured Storage System," *SIGOPS Operating Systems Review*, vol. 44, 2010.

[20] G. DeCandia, D. Hastorun, M. Jampani *et al.*, "Dynamo: amazon's Highly Available Key-Value Store," in *Proc. ACM SOSP*, 2007.

[21] J. Baker, C. Bondç, J. C. Corbett *et al.*, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *Proc. CIDR*, 2011.

[22] J. Rao, E. J. Shekita, and S. Tata, "Using Paxos to build a Scalable, Consistent, and Highly Available Datastore," *Proceedings of the VLDB Endowment*, vol. 4, 2011.