Achieving Robust Self-Management for Large-Scale Distributed Applications

Ahmad Al-Shishtawy*†, Muhammad Asif Fayyaz*, Konstantin Popov†, and Vladimir Vlassov*

*Royal Institute of Technology, Stockholm, Sweden
{ahmadas, mafayyaz, vladv}@kth.se
†Swedish Institute of Computer Science, Stockholm, Sweden
{ahmad, kost}@sics.se

***Abstract*—Achieving self-management can be challenging, particularly in dynamic environments with resource churn (joins/leaves/failures). Dealing with the effect of churn on management increases the complexity of the management logic and thus makes its development time consuming and error prone. We propose the abstraction of robust management elements (RMEs), which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of dealing with the effect of churn on management from the management logic. This facilitates the development of robust management by making the developer focus on managing the application while relying on the platform to provide the robustness of management. RMEs can be implemented as fault-tolerant long-living services.**

We present a generic approach and an associated algorithm to achieve fault-tolerant long-living services. Our approach is based on replicating a service using finite state machine replication with a reconfigurable replica set. Our algorithm automates the reconfiguration (migration) of the replica set in order to tolerate continuous churn. The algorithm uses P2P replica placement schemes to place replicas and uses the P2P overlay to monitor them. The replicated state machine is extended to analyze monitoring data in order to decide on when and where to migrate. We describe how to use our approach to achieve robust management elements. We present a simulation-based evaluation of our approach which shows its feasibility.

***Keywords*-autonomic computing; distributed systems; self-management; replicated state machines; service migration; P2P.**

I. Introduction

Autonomic computing [1] is a paradigm to deal with management overhead of complex systems by making them self-managing. Self-management can be achieved through autonomic managers [2] that monitor the system and act accordingly. In our previous work, we have developed a platform called Niche [3], [4] that enables one to build self-managing large-scale distributed systems. An autonomic manager in Niche consists of a network of management elements (MEs) each of which can be responsible for one or more roles of the MAPE loop [2]: Monitor, Analyze, Plan, and Execute. MEs are distributed and interact with each other through events.

Large-scale distributed systems are typically dynamic with resources that may fail, join, or leave the system at any time (resource churn). Constructing autonomic managers in environments with high resource churn is challenging because MEs need to be restored with minimal disruption to the autonomic manager, whenever the resource (where an ME executes) leaves or fails. This challenge increases if the MEs are stateful because the state needs to be maintained.

We propose the Robust Management Element (RME) abstraction that allows simplifying the development of robust autonomic managers that can tolerate resource churn, and thus self-managing large-scale distributed systems. With RMEs, developers of self-managing systems can assume that management elements never fail. An RME 1) is replicated to ensure fault-tolerance; 2) tolerates continuous churn by automatically restoring failed replicas on other nodes; 3) maintains its state consistent among replicas; 4) provides its service with minimal disruption in spite of resource churn (high availability), and 5) is location transparent, i.e. RME clients communicate with it regardless of current location of its replicas. Because we target large-scale distributed environments with no central control, all algorithms of the RME abstraction should be decentralized.

In this paper, we present our approach to implement RMEs which is based on state machine replication [5] combined with *automatic* reconfiguration of replica set. Replication by itself is insufficient to guarantee long-term fault-tolerance under continuous churn, as the number of failed nodes hosting ME replicas, and hence the number of failed replicas, will increase over time, and eventually RME will stop. Therefore, we use *service migration* [6] to enable the reconfiguration of the set of nodes hosting ME replicas. Using service migration, new nodes can be introduced to replace the failed ones. We propose a decentralized algorithm that will use migration to *automatically* reconfigure the set of nodes hosting ME replicas. This will guarantee that RME will tolerate continuous churn.

The major contributions of this paper are:

- The use of *Structured Overlay Networks* (SONs) [7] to monitor the nodes hosting replicas in order to detect changes that may require reconfiguration. SONs are also used to determine replica location using replica placement schemes such as symmetric replication [8].
- The replicated state machine, beside replicating a service, receives monitoring information and uses it to construct a new configuration and to decide when to migrate.
- A decentralized algorithm that automates the reconfiguration of the replica set in order to tolerate continuous resource churn.

The remainder of the paper is organised as follows. Section II presents necessary background. In Section III we describe our decentralized algorithm to automate the reconfiguration process. Section IV describes how our approach can be applied to achieve RMEs in Niche. In Section V we discuss our experimental results. Related work is discussed in Section VI. Finally, Section VII presents conclusions and our future work.

II. Background

This section presents the necessary background to our approach and algorithms presented in this paper, namely: The Niche platform, SON and symmetric replication, replicated state machines, and an approach to migrate stateful services.

A. Niche Platform

Niche [3] is a distributed component management system that implements the autonomic computing architecture [2]. Niche includes a programming model, APIs, and a runtime system. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed in a dynamic distributed environment where resources can join, leave, or fail. A self-managing application in Niche consists of functional and management parts. Functional components communicate via interface bindings, whereas management components communicate via a publish/subscribe event notification mechanism.

The Niche runtime environment is a network of containers hosting functional and management components. Niche uses a Chord [7]-like structured overlay network (SON) as its communication layer. The SON is self-organising on its own and provides overlay services such as address lookup, Distributed Hash Table (DHT) and a publish/subscribe mechanism for event dissemination. Niche provides higher-level communication abstractions such as name-based bindings to support component mobility, dynamic component groups, one-to-any and one-to-all group bindings, and event based communication.

B. Structured Overlay Networks and Symmetric Replication

Structured Overlay Networks (SONs) are known for their self-organisation and resilience under churn [9]. We assume the following model of SONs and their APIs. In the model, SON provides the `lookup` operation to locate items on the network. For example, items can be data items for DHTs, or some compute facilities that are hosted on individual nodes in a SON. We say that the node hosting or providing access to an item is responsible for that item. Both items and nodes posses unique SON identifiers that are assigned from the same identifier space. The SON automatically and dynamically divides the responsibility between nodes such that for every SON identifier there is always a responsible node. The `lookup` operation returns the address of a node responsible for a given SON identifier. Because of churn, node responsibilities change over time and, thus, `lookup` can return over time different nodes for the same item. In practical SONs, the `lookup` operation can also occasionally return wrong (inconsistent) results due to churn. Furthermore, SON can notify application software running on a node when the responsibility range of the node changes. When responsibility changes, items need to be moved between nodes accordingly.

In Chord-like SONs the identifier space is circular, every node is responsible for items with identifiers in the range between the identifier of its predecessor and its own identifier. Such a SON naturally provides for symmetric replication of items on the SON, where replica identifiers are placed symmetrically around the identifier space circle.

Symmetric Replication [8] is a scheme used to determine replica placement in SONs. Given an item identifier i, a replication degree f, and the size of the identifier space N, symmetric replication is used to calculate the identifiers of the item's replicas. The identifier of the x-th $(1 \leq x \leq f)$ replica of the item i is computed as follows:

$$r(i, x) = (i + (x - 1)N/f) \bmod N \qquad (1)$$

C. Replicated State Machines

A common way to achieve high availability of a service is to replicate it on several nodes. Replicating stateless services is relatively simple and is not considered here. A common way to replicate stateful services is to use the replicated state machine approach [10]. Using this approach requires the service to be deterministic. A set of deterministic services will have the same state change and produce the same output given the same sequence of inputs and the same initial state. This implies that sources of nondeterminism, such as local clocks, random numbers, and multi-threading, should be avoided.

Replicated state machines can use the Paxos [11] consensus algorithm to ensure that all service replicas execute the same input requests in the same order. Paxos relies on a leader election algorithm, such as [12], to elect one of the replicas as the leader. The leader determines the order of requests by proposing slot numbers for requests. Paxos assigns requests to slots. Several requests can be processed by Paxos concurrently. Replicas execute an assigned request after all requests assigned to previous slots have been executed. Paxos can tolerate replica failures and still operate correctly as long as the number of failures is less than half of the total number of replicas.

D. Migrating Stateful Services

SMART [6] is a technique for changing the set of nodes where a replicated state machine runs, i.e. for migrating the service to a new set of nodes. A fixed set of nodes, where a replicated state machine runs, is called a *configuration*.

SMART is built on the migration technique proposed by Lamport [11] where the configuration is kept as a part of the service state. Migration to a new configuration proceeds by executing a special state-change request that describes the configuration change. Lamport also proposed to delay the effect of the configuration change (i.e., using the new configuration) for α slots after the state-change request have been executed. This improves performance by allowing to assign concurrently α more requests in the current configuration.

SMART provides a complete treatment of Lamport's idea, but it does not provide a specific algorithm for automatic configuration management. SMART also allows to replace non-failed nodes, enabling configuration management that occasionally removes working nodes due to, e.g., an imperfect failure detector.

The central idea in SMART is the configuration-specific replicas. SMART preforms service migration from a configuration `conf1` to a new configuration `conf2` by creating a new independent set of replicas for `conf2` that run, for a while, in parallel with replicas in `conf1`. The first slot of `conf2` is assigned to be the next slot after the last slot of `conf1`. The replicas in `conf1` are kept long enough to ensure that `conf2` is established and replica state is transferred to new nodes. This simplifies the migration process and helps SMART to overcome limitations of other techniques. Nodes that carry replicas in both `conf1` and `conf2` keep a single copy of replica state per node. The state shared by replicas of different configurations is maintained by a so-called *execution module*. Each configuration runs its own instance of the Paxos algorithm independently without sharing. Thus, from the point of view of the replicated state machine instance, it looks like as if the Paxos algorithm is running on a static configuration.

III. Automatic Reconfiguration of Replica Sets

In this section we present our approach and associated algorithms to achieve robust services. Our approach automates the process of selecting a replica set (configuration) and the decision of migrating to a new configuration in order to provide a robust service that can tolerate continuous resource churn and run for long periods of time without the need of human intervention. The approach uses the replicated state machine technique, migration support, and the symmetric replication scheme. Our approach was mainly designed to provide the Robust Management Elements (RMEs) abstraction which is used to achieve robust self-management. An example is our platform Niche [3], [4] where this approach can be applied directly and RMEs can be used to build robust autonomic managers. However, we believe that our approach is generic enough to be used to achieve other robust services.

We assume that the environment that will host the Replicated State Machines (RSMs) consists of a number of nodes forming a Structured Overlay Network (SON) that may host multiple RSMs. Each RSM is identified by a constant ID (denoted RSMID) drawn from the SON identifier space. RSMID permanently identifies an RSM regardless of the number of nodes in the system and node churn that causes reconfiguration of the replica set. Given an RSMID and the replication degree, the symmetric replication scheme is used to calculate the SON ID of each replica. The replica SON ID determines the node responsible for hosting the replica. This responsibility, unlike the replica ID, is not fixed and may changes over time due to churn. Clients that send requests to the RSM need to know only its RSMID and replication degree. With this information clients can calculate identifiers of individual replicas using the symmetric replication scheme, and locate the nodes currently responsible for the replicas using the lookup operation provided by the SON. Most of the nodes found in this way will indeed host the RSM replicas, but not necessarily all of them because of lookup inconsistency and churn.

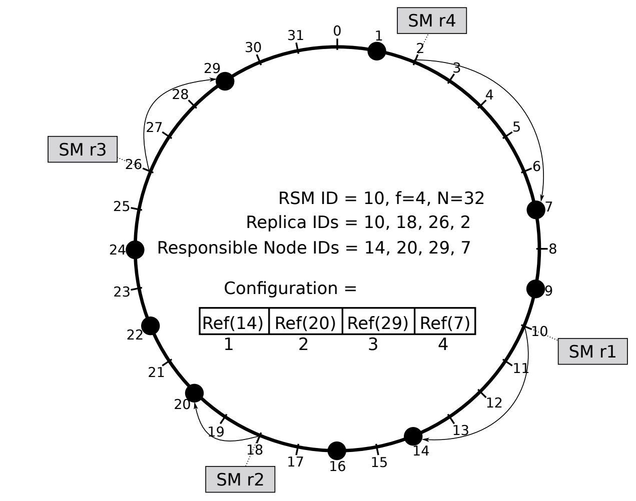
Fault-tolerant consensus algorithms like Paxos require a fixed set of known replicas that we call configuration. Some of replicas, though, can be temporarily unreachable or down (the crash-recovery model). The SMART protocol extends the Paxos algorithm to enable explicit reconfiguration of replica sets. Note that RSMIDs cannot be used for neither of the algorithms because the lookup operation can return over time different sets of nodes. In the algorithm we contribute for management of replica sets, individual RSM replicas are mutually identified by their addresses which in particular do not change under churn. Every single replica in a RSM configuration knows addresses of all other replicas in the RSM.

The RSM, its clients and the replica set management algorithm work roughly as follows. A dedicated initiator chooses RSMID, performs lookups of nodes responsible for individual replicas and sends to them a request to create RSM replicas. Note the request contains RSMID, replication degree, and the configuration consisting of all replica addresses, thus newly created replicas perceive each other as a group and can communicate with each other directly withoud relying on the SON. RSMID is also distributed to future RSM clients.

Because of churn, the set of nodes responsible for individual RSM replicas changes over time. In response, our distributed configuration management algorithm creates new replicas on nodes that become responsible for RSM replicas, and eventually deletes unused ones. The algorithm consists of two main parts. The first part runs on all nodes of the overlay and is responsible for monitoring and detecting changes in the replica set caused by churn. This part uses several sources of events and information, including SON node failure notifications, SON notifications about change of responsibility, and requests from clients that indicates the absence of a replica. Changes in the replica set (e.g. failure of a node that hosted a replica) will result in a *configuration change request* that is sent to the corresponding RSM. The second part is a special module, called the *management module*, that is dedicated to receive and process monitoring information (the configuration change requests). The module use this information to construct a configuration and also to decide when it is time to migrate (after a predefined number of changes in the configuration). We discuss the algorithm in greater detail in the following.

A. Configurations and Replica Placement Schemes

All nodes in the system are part of SON as shown in Fig. 1. The RSM that represents the service is assigned an $RSMID$ from the SON identifier space of size N. The set of nodes that will form a configuration are selected using the symmetric replication scheme [8]. The symmetric replication, given the replication factor f and the $RSMID$, is used to calculate the *Replica IDs* according to equation 1. Using the `lookup()` operation, provided by the SON, we can obtain the IDs and direct references (IP address and port) of the responsible nodes. These operations are shown in Algorithm 1. The rank of a replica is the parameter x in equation 1. A



SM r4
SM r3
SM r1
SM r2
RSM ID = 10, f=4, N=32
Replica IDs = 10, 18, 26, 2
Responsible Node IDs = 14, 20, 29, 7
Configuration =
Ref(14) Ref(20) Ref(29) Ref(7)
1 2 3 4

Fig. 1. Replica Placement Example: Replicas are selected according to the symmetric replication scheme. A Replica is hosted (executed) by the node responsible for its ID (shown by the arrows). A configuration is a fixed set of direct references (IP address and port) to nodes that hosted the replicas at the time of configuration creation. The RSM ID and Replica IDs are fixed and do not change for the entire life time of the service. The Hosted Node IDs and Configuration are only fixed for a single configuration. Black circles represent physical nodes in the system.

Algorithm 1 Helper Procedures

```
1: procedure GetConf($RSMID$)
2:     $ids[\,] \leftarrow$ GetReplicaIDs($RSMID$)                    ▷ Replica Item IDs
3:     for $i \leftarrow 1, f$ do $refs[i] \leftarrow$ Lookup($ids[i]$)
4:     end for
5:     return $refs[\,]$
6: end procedure

7: procedure GetReplicaIDs($RSMID$)
8:     for $x \leftarrow 1, f$ do $ids[x] \leftarrow$ r($RSMID, x$)          ▷ See equation 1
9:     end for
10:    return $ids[\,]$
11: end procedure
```

configuration is represented by an array of size f. The array holds direct *references* (IP and port) to the nodes that form the configuration. The array is indexed from 1 to f, and each element contains the reference to the replica with the corresponding rank.

The use of direct references, instead of using lookup operations, as the configuration is important for our approach to work for two reasons. First reason is that we can not rely on the lookup operation because of the lookup inconsistency problem. The lookup operation, used to find the node responsible for an ID, may return incorrect references. These incorrect references will have the same effect in the replication algorithm as node failures even though the nodes might be alive. Thus the incorrect references will reduce the fault tolerance of the replication service. Second reason is that the migration algorithm requires that both the new and the previous configurations coexist until the new configuration is established. Relying on lookup operation for `replica_IDs` may not be possible. For example, in Fig. 1, when a node with $ID = 5$ joins the overlay it becomes responsible for the replica `SM_r4` with $ID = 2$.

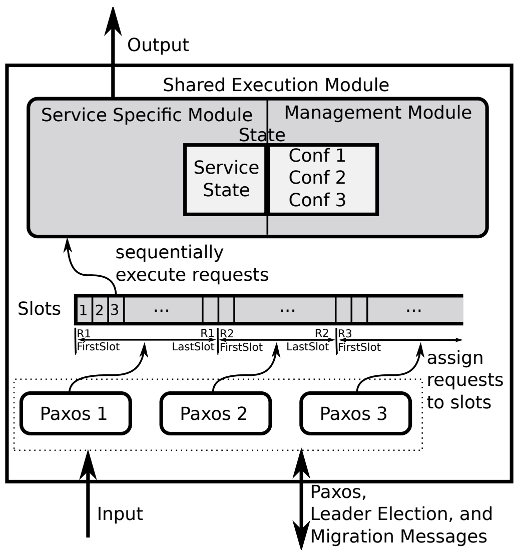


Fig. 2. State Machine Architecture: Each machine can participate in more than one configuration. A new replica instance is assigned to each configuration. Each configuration is responsible for assigning requests to a none overlapping range of slot. The execution module executes requests sequentially that can change the state and/or produce output.

A correct `lookup(2)` will always return 5. Because of this, the node 7, from the previous configuration, will never be reached using the lookup operation. This can also reduce the fault tolerance of the service and prevent the migration in the case of large number of joins.

Nodes in the system may join, leave, or fail at any time (churn). According to the Paxos, a configuration can survive the failure of less than half of the nodes in the configuration. In other words, $f/2+1$ nodes must be alive for the algorithm to work. This must hold independently for each configuration. After a new configuration is established, it is safe to destroy instances of older configurations.

Due to churn, the responsible node for a certain replica may change. For example in Fig.1 if node 20 fails then node 22 becomes responsible for identifier 18 and should host `SM_r2`. The algorithms described below automate the migration process by detecting the change and triggering a `ConfChange` request. The `ConfChange` request will be handled by the state machine and will eventually cause it to migrate to a new configuration.

B. State Machine Architecture

The replicated state machine (RSM) consists of a set of replicas, which forms a configuration. Migration techniques can be used to change the configuration. The architecture of a replica, shown Fig. 2, uses the shared execution module optimization presented in [6]. This optimization is useful when the same replica participates in multiple configurations. The execution module executes requests. The execution of a request may result in state change, producing output, or both. The execution module should be deterministic. Its outputs and states must depend only on the sequence of input and the initial state. The execution module is also required to support checkpointing which enables state transfer between replicas.

The execution module is divided into two parts: the service specific module and the management module. The service specific module captures the logic of the service and executes all requests except the `ConfChange` request which is handled by the management module. The management module maintains a *next configuration* array that it uses to store `ConfChange` requests in the element with the corresponding rank. After a predefined threshold of the number and type (join/leave/failure) of changes, the management module decides that it is time to migrate. It uses the next configuration array to update the current configuration array resulting in a new configuration. After that, the management module passes the new configuration to the migration protocol to actually preform the migration. The reason to split the state into two parts is because the management module is generic and independent of the service and can be reused with different services. This simplifies the development of the service specific module and makes it independent from the replication technique. In this way legacy services, that are already developed, can be replicated without modification given that they satisfy execution module constraints (determinism and checkpointing).

In a corresponding way, the state of a replica consists of two parts: The first part is internal state of the service specific module which is application specific; The second part consists of the configurations. The remaining parts of the replica, other than the execution module, are responsible to run the replicated state machine algorithms (Paxos and Leader Election) and the migration algorithm (SMART). As described in the previous section, each configuration is assigned a separate instance of the replicated state machine algorithms. The migration algorithm is responsible for specifying the `FirstSlot` and `LastSlot` for each configuration, starting new configurations, and destroying old configurations after the new configuration is established.

The Paxos algorithm guarantees liveness when a single node acts as a leader, thus it relies on a fault-tolerant leader election algorithm. Our system uses the algorithm described in [12]. This algorithm guarantees progress as long as one of the participating processes can send messages such that every message obtains f timely (i.e. with a pre-defined timeout) responses, where f is a algorithm's constant parameter specifying how many processes are allowed to fail. Note that the f responders may change from one algorithm round to another. This is exactly the same condition on the underlying network that a leader in the Paxos itself relies on for reaching timely consensus. Furthermore, the aforementioned work proposes an extension of the protocol aiming to improve leader stability so that qualified leaders are not arbitrarily demoted which causes significant performance penalty for the Paxos protocol.

C. Replicated State Machine Maintenance

This section describes the algorithms used to create a replicated state machine and to automate the migration process in order to survive resource churn.

Algorithm 2 Replicated State Machine API

```
 1: procedure CreateRSM($RSMID$)
                                ▷ Creates a new replicated state machine
 2:     $Conf[\,] \leftarrow$ GetConf($RSMID$)
                                ▷ Hosting Node REFs
 3:     for $i \leftarrow 1, f$ do
 4:         sendto $Conf[i]$ : InitSM($RSMID, i, Conf$)
 5:     end for
 6: end procedure

 7: procedure JoinRSM($RSMID, rank$)
 8:     SubmitReq($RSMID, ConfChange(rank, MyRef)$)
          ▷ The new configuration will be submitted and assigned a slot to be executed
 9: end procedure

10: procedure SubmitReq($RSMID, req$)
                                ▷ Used by clients to submit requests
11:     $Conf[\,] \leftarrow$ GetConf($RSMID$)
                                ▷ $Conf$ is from the view of the requesting node
12:     for $i \leftarrow 1, f$ do
13:         sendto $Conf[i]$ : Submit($RSMID, i, Req$)
14:     end for
15: end procedure
```

1) State Machine Creation: A new RSM can be created by any node by calling `CreateRSM` in Algorithm 2. The creating node constructs the configuration using symmetric replication and lookup operations. The node then sends an `InitSM` message to all nodes in the configuration. Any node that receives the message (Algorithm 5) starts a state machine (SM) regardless of its responsibility. Note that the initial configuration, due to lookup inconsistency, may contain some incorrect references. This does not cause problems for the RSM because all incorrect references in the configuration will eventually be detected and corrected by our algorithms.

2) Client Interactions: A client that requires the service provided by the RSM can be on any node in the system. The client needs to know only the $RSMID$ and the replication degree to be able to send requests to the service. Knowing the $RSMID$, the client can determine the current configuration using equation 1 and lookup operations (See Algorithm 1). In this way we avoid the need for an external configuration repository that points to nodes hosting the replicas in the current configuration. The client submits requests by calling `SubmitReq`, shown in Algorithm 2, that sends the request to all replicas in the current configuration. Due to lookup inconsistency, that can happen either at the client side or the RSM side, the client's view of the configuration and the actual configuration may differ. For the client to be able to submit requests, the client's view must overlap, at least at one node, with the actual configuration. Otherwise, the request will fail and the client can retry later. We assume that each request is uniquely stamped and that duplicate requests are filtered. In the current algorithm the client submits the request to all nodes in the configuration. It is possible to optimise the number of messages by submitting the request only to one node in the configuration that will forward it to the current leader. The trade off is that sending to all nodes increases the probability of the request reaching the RSM. This reduces the negative effects of lookup inconsistencies and churn on the availability of the service. Clients may also cache the reference to the current leader and use it directly until the leader changes.

Algorithm 3 Execution

```
1: receipt of SUBMIT($RSMID, rank, Req$) from m at n
2:     $SM \leftarrow SMs[RSMID][rank]$
3:     if $SM \neq \phi$ then                              ▷ Node is hosting the replica
4:         if $SM.leader = n$ then $SM.schedule(Req)$      ▷ Paxos schedule it
5:         else                                  ▷ forward the request to the leader
6:             sendto $SM.leader$ : SUBMIT($RSMID, rank, Req$)
7:         end if
8:     else                                            ▷ Node is not hosting the replica
9:         if $r(RSMID, rank) \in ]n.predecessor, n]$ then     ▷ I'm responsible
10:            JOINRSM($RSMID, rank$)                      ▷ Fix the configuration
11:        else                                            ▷ I'm not responsible
12:            DONOTHING              ▷ This is probably due to lookup inconsistency
13:        end if
14:    end if
15: end receipt

16: procedure EXECUTESLOT(req)                              ▷ The Execution Module
17:     if req.type = ConfChange then                      ▷ The Management Module
18:         $nextConf[req.rank] \leftarrow req.id$
                                  ▷ Update the candidate for the next configuration
19:         if $nextConf.changes = threshold$ then
20:             $newConf \leftarrow$ UPDATE(CurrentConf,NextConf)
21:             $SM.migrate(newConf)$
                                  ▷ SMART will set LastSlot and start new configuration
22:         end if
23:     else                    ▷ The Service Specific Module handles all other requests
24:         $ServiceSpecificModule.Execute(req)$
25:     end if
26: end procedure
```

3) Request Execution: The execution of client requests is initiated by receiving a submit request from a client and consists of three steps: checking if the node is responsible for the RSMID in the request, scheduling the request, and executing it. These steps are shown in Algorithm 3.

When a node receives a request from a client it will first check, using the RSMID in the request, if it is hosting the replica to which the request is directed to. If this is the case, then the node will submit the request to that replica. The replica will try to schedule the request for execution if the replica believes that it is the leader. Otherwise the replica will forward the request to the leader. The scheduling is done by assigning the request to a slot that is agreed upon among all replicas in the configuration (using the Paxos algorithm). Meanwhile, the execution module executes scheduled requests sequentially in the order of their slot numbers.

On the other hand, if the node is not hosting a replica with the corresponding RSMID, it will proceed with one of the following two scenarios. In the first scenario, it may happen due to lookup inconsistency that the configuration calculated by the client contains some incorrect references. In this case, a incorrectly referenced node ignores client requests (Algorithm 3 line 12) because it is not responsible for the target RSM. In the second scenario, it is possible that the client's view is correct but the current configuration contains some incorrect references. In this case, the node that discovers, through the client request, that it was supposed to be hosting a replica will attempt to correct the current configuration by sending a `ConfChange` request replacing the incorrect reference with the reference to itself (Algorithm 3 line 10). At execution time, the execution module will direct all requests except the `ConfChange` request to the service specific module for execution. The `ConfChange` will be directed to the management module for processing.

Algorithm 4 Churn Handling

```
1: procedure NODEJOIN           ▷ Called by SON after the node joined the overlay
2:     sendto $successor$ : PULLSMS($]predecessor, myId]$)
3: end procedure

4: procedure NODELEAVE
   sendto $successor$ : NEWSMS(SMs)       ▷ Transfer all hosted SMs to Successor
5: end procedure

6: procedure NODEFAILURE($newPredID, oldPredID$)
                                       ▷ Called by SON when the predecessor fails
7:     $I \leftarrow \bigcup_{x=2}^{f} ]r(newPredID, x), r(oldPredID, x)]$
8:     multicast $I$ : PULLSMS($I$)
9: end procedure
```

Algorithm 5 SM maintenance (handled by the container)

```
1: receipt of INITSM($RSMID, Rank, Conf$) from m at n
2:     new $SM$                        ▷ Creates a new replica of the state machine
3:     $SM.ID \leftarrow RSMID$
4:     $SM.Rank \leftarrow Rank$                              ▷ $1 \leq Rank \leq f$
5:     $SMs[RSMID][Rank] \leftarrow SM$  ▷ SMs stores all SM that node n is hosting
6:     $SM.Start(Conf)$                      ▷ This will start the SMART protocol
7: end receipt

8: receipt of PULLSMS($Intervals$) from m at n
9:     for each $SM$ in $SMs$ do
10:        if R($SM.id, SM.rank$) $\in I$ then
11:            $newSMs.add(SM)$
12:        end if
13:    end for
14:    sendto m : NEWSMS($newSMs$)
15: end receipt

16: receipt of NEWSMS($NewSMs$) from m at n
17:     for each $SM$ in $NewSMs$ do
18:         JOINRSM($SM.id, SM.rank$)
19:     end for
20: end receipt
```

4) Handling Churn: Algorithm 4 contains procedures to maintain the replicated state machine when a node joins, leaves, or fails. When any of these events occur, a new node might become responsible for hosting a replica. In the case of node join, the new node will send a message to its successor to get information (RSMID and replication degree) about any replicas that the new node should be responsible for. In the case of leave, the leaving node will send a message to its successor containing information about all replicas that it was hosting. In the case of failure, the successor of the failed node needs to discover if the failed node was hosting any replicas. This can be done in a proactive way by checking all intervals (Algorithm 4 line 7) that are symmetric to the interval that the failed node was responsible for. One way to achieve this is by using range-cast that can be efficiently implemented on SONs, e.g., using bulk operations [8]. The discovery can also be done lazily using client requests as described in the previous section and Algorithm 3 line 10.

In all three cases described above, newly discovered replicas are handled by `NewSMs` (Algorithm 5). The node will request a configuration change by joining the corresponding RSM for each new replica. Note that the configuration size is fixed to f. A configuration change means replacing reference at position r in the configuration array with the reference of the node requesting the change.

IV. Robust Management Elements in Niche

The proposed approach with corresponding algorithms, described in the previous section, allows meeting the requirements of the Robust Management Element (RME) abstraction specified in Section I. It can be used to implement the RME abstraction in Niche in order to achieve robustness and high availability of autonomic managers in spite of churn. An autonomic manager in Niche is constructed from a set of management elements (MEs). A robust management element can be implemented by wrapping an ordinary ME inside a state machine which is transparantly replicated by the RME support added to the Niche platform. The ME will serve as the service-specific module shown in Fig. 2. However, to be able to use this approach, the ME must follow the same constraints as the execution module, that is the ME must be deterministic and provide checkpointing. The clients (e.g., sensors) need only to know the RME identifier to be able to use an RME regardless of the location of individual replicas. The RME support in the Niche platform will facilitate development of applications with robust self-management.

V. Prototype and Evaluation

In this section, we present a simulation-based performance evaluation of our approach for replicating and maintaining stateful services in various scenarios. In evaluating the performance, we are mainly interested in measuring the *request latency* and the *number of messages* exchanged by our algorithms. The evaluation is divided in three main categories: critical path evaluation, failure recovery evaluation, and evaluation of the overheads associated with the leader election.

To evaluate the performance of our approach and to show the practicality of our algorithms, we built a prototype implementation of Robust Management Elements using the *Kompics* component model [13]. Kompics is a framework for building and evaluating distributed systems in simulation, local execution, and distributed deployments. In order to make network simulation more realistic, we used the King latency dataset, available at [14], that measures the latencies between DNS servers using the King [15] technique. For the underlying SON, we used Chord implementation provided by Kompics. To evaluate the performance of our algorithms in various churn scenarios, we have used the *lifetime-based node failure* model [9], [16] with the shifted Pareto lifetime Distribution.

A. Methodology

In the simulation scenarios described below, we assumed one stateful service (a Robust Management Element) and several clients (sensors and actuators). A client represents both a sensor and an actuator. The service is replicated using our approach. For simplicity but without losing generality, the service is implemented as an aggregator that accumulates integer values received from clients and replies with the current aggregated value which is the state of the service. A client request (containing a value) represents monitoring information whereas a service response represents an actuation command. Each client repeatedly sends requests to the service. Upon receiving a client request, the service performs all the actions related to the replicated state machine, makes a state transition, and sends the response to the requesting client.

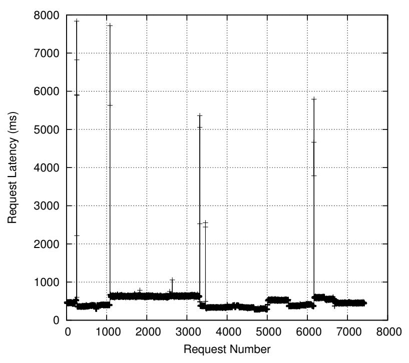
There are various factors in a dynamic distribution environment that can influence the performance of our approach. The input parameters to the simulator include:

- Numeric (architectural) parameters:
  - Overlay size: in the rage of 200 to 600 nodes;
  - Number of services (management elements): 1;
  - Number of clients: 4;
  - Replication degree: varies from 5 to 25;
  - Failure threshold: this is the number of failures that will cause the RSM to migrate. This can range from 1 to strictly less than half of the number of replicas.
- Timing (operational) parameters
  - Shifted Pareto distribution of client requests with a specified mean time between consecutive requests. In the simulations we used four clients each with mean time between requests of 4 seconds. This gives the total mean time of 1 second between requests from all four clients to the service.
  - Shifted Pareto distribution of node life time with a specified mean to model churn. We modeled three levels of churn: high churn rate (mean life time of 30 minutes), medium churn rate (90 minutes), low churn rate (150 minutes).

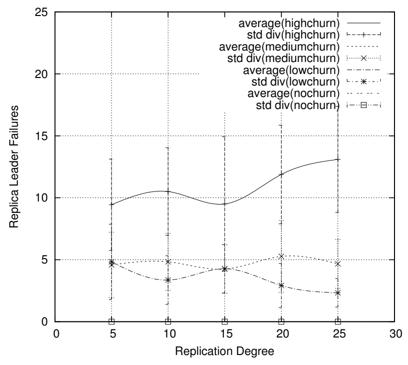
In our experiments, we have enabled pipelining of requests (by setting α to 10) as suggested by SMART [6], i.e., up to 10 client requests can be handled concurrently. In all plots, unless otherwise stated, we simulated 8 hours. The plot is the average of 10 independent runs with standard deviation bars.

In our simulation, we have evaluated how the performance of the proposed algorithms depends on the replication degree (the number of replicas) and the overlay size (the number of physical nodes). The overlay size affects the performance (time and message complexity) of overlay operations [8], namely, lookup and range-cast, used by our algorithms in the following three cases: (1) when creating the initial RSM configuration that is done only once, (2) when looking up the current configuration, and (3) when performing failure recovery. The intensity of configuration lookups depends on the churn rate, and it can be reduced by caching the lookup results (configuration). The intensity of failure recovery depends on the failure rate. Therefore, if the rate of churn (including failures) is lower than the client request rate, the performance of our approach mostly depends on the replication degree rather than on the overlay size. This is because the overlay operations happen relatively rarely. With increasing the overlay size, we expect our approach to scale due to the logarithmic scalability of the overlay operations.

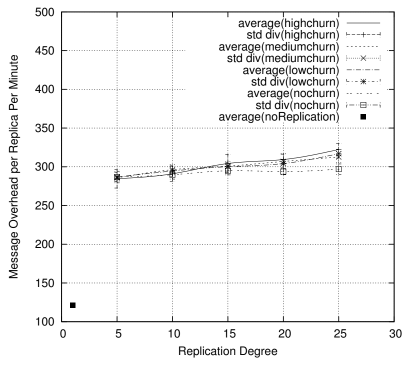
In our experiments, we assumed a reasonable load (the request rate) on the system, and churn rates which are lower than the client request rate. We simulated overlays with hundreds of nodes. Study of systems with larger scales and/or extreme values of load and churn rates is in our future work.



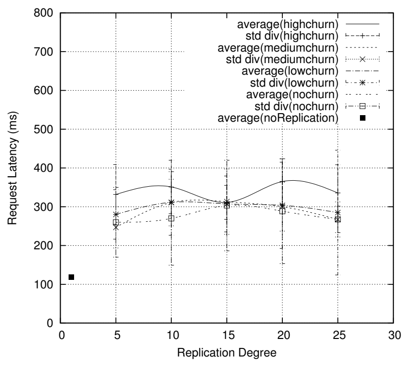
(a) Request latency for a single client



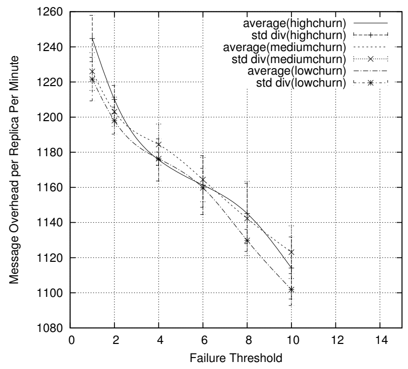
(b) Leader failures vs. replication degree



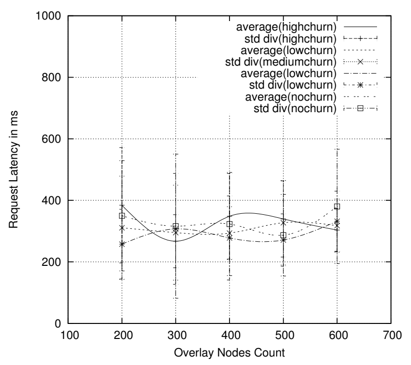
(c) Messages/minute vs. replication degree



(d) Request latency vs. replication degree



(e) Messages per minute vs. failure threshold



(f) Request latency vs. overlay size

Fig. 3. Request latency and message overhead

We use more than one client in order to get a better average of client-server communication latency. The mean time between requests of 1 sec was selected (via testing experiments) as a reasonable workload that does not overload the system.

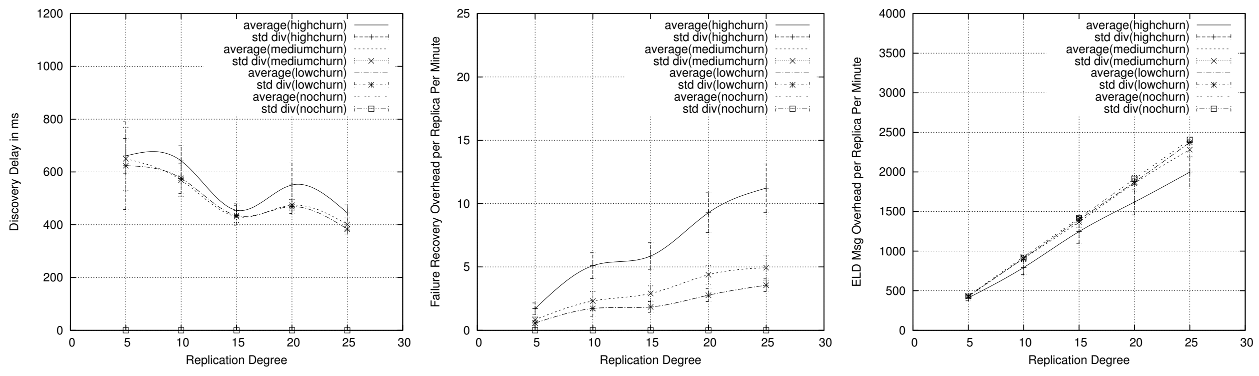
The baseline in our evaluation is the system with no replication and no churn. We expect that the baseline system has better performance compared to the performance of a system with replication and with/without churn. This is because the replication mechanism as well as migration caused by churn introduce overhead. There are three kinds of overhead in the system: (i) Paxos (which happens upon arrival of requests to RSM), (ii) RSM migration (which happens upon churn), and (iii) the leader election algorithm (which runs continuously). All the overheads cause increase in the number of messages and may cause performance degradation. In our experiments described below we compare performance of different system configurations (overlay size and replication degree) and different churn rates against the baseline system configuration with no replication and no churn (hence no migration).

B. Simulation Scenarios

1) Request Critical Path: In this series of experiments we study the effect of various input parameters on the performance (the request latency and the number of messages) of handling client requests. The request critical path includes sending the request, Paxos, migration, and receiving the reply.

The effect of churn on performance (request latency) is minimal. Fig. 3(a) depicts latencies of requests submitted by a single client during 8 hours in a system with a high churn rate. Out of more than 7000 requests, only less than 20 requests were severely affected by churn. The spikes happen when the leader in the Paxos algorithm fails. This is because Paxos can not proceed until a new leader is elected. The average number of leader failures during 8 hours is shown in Fig. 3(b). This can help to estimate the number of such spikes that can happen in a system with a specified replication degree and churn rate. The time to detect the failure of the current leader and elect a new leader is maximum 10 seconds according to the leader election parameters used in the simulations. During this time any request that arrives at the RSM will be delayed. If non-leader fails, the RSM is not affected as long as the total number of failed replicas is less than the failure threshold parameter. If the number of failed replicas is at least the value of the failure threshold parameter then a migration will happen. On average a migration takes 300 milliseconds to complete.

Using our approach increases the number of critical path messages needed to handle a request compared to the number of messages in the baseline (Fig. 3(c)). This is mainly because of the Paxos messages needed to reach consensus for every request. However, increasing the replication degree does not significantly increase the number of messages per replica, as



(a) Discovery delay vs. replication degree (b) Recovery messages vs. replication degree (c) Leader election overhead

Fig. 4. Failure recovery and leader election

can be seen in Fig. 3(c). The slight increase in the number of messages, when the replication degree increases, is due to the increase in the number of migrations. The number of messages is also affected by the churn rate. This is because the higher the churn rate is the higher the migration rate will be. On the other hand, the request latency, as shown in Fig. 3(d), is not affected by the replication degree because Paxos requires two phases regardless of the number of replicas. Fig. 3(d) also shows the overhead of our approach compared to the baseline.

The average number of critical path messages per request is also affected by the failure threshold parameter as shown in Fig. 3(e). A higher failure threshold results in the lower number of messages caused by migration. This is because the higher the failure threshold is, the lower the migration rate will be. For example, with the threshold of 1, the RSM will migrate immediately after one failure; whereas with the threshold of 10, it will wait for 10 replicas to fail before migrating. In this experiment we used 25 replicas. Note that in this case the maximum possible failure threshold is 12. In order to highlight the effect of failure threshold on the message complexity, we increased the request rate from 1 to 4 requests per second.

Our experiments for overlays with hundreds of nodes have shown that the overlay size has minimal or no impact on the request latency, as depicted in Fig. 3(f). The request latency deviates when changing the overlay size. One of the possible explanations could be a possible deviations in the average communication latency due to the use of the King latency dataset for network delay. This requires further study and more simulation experiments.

In the above experiments we did not include the lookup performed by clients to discover the configuration. This is because it is not on the critical path, as clients may cache the configuration. For this reason the performance is not affected by the overlay size because all critical path messages are passed over direct links rather that through the overlay.

2) Failure Recovery: When an overlay node fails, another node (the successor) becomes responsible for any replicas hosted by the failed node. The successor node needs to discover if any replicas were hosted on the failed node. In the simulation experiments we used overlay range-cast to do the discovery. Note that this process is not on the critical path for processing client requests since both can happen in parallel.

Fig. 4(a) depicts the discovery delay for various replication degrees. The discovery delay decreases when the number of replicas increases. This is because it is enough to discover only one replica, and it takes shorter time to find a replica in a system with a higher replication degree, as the probability to find a replica which is close (in terms of link latency and/or overlay hops) to the requesting node is higher. As shown in Fig. 4(b), a higher churn rate requires more failure recovery and thus causes higher message overhead.

3) Other Overheads: Maintaining the SON introduces an overhead in term of messages. We did not count these messages in our evaluation because they vary a lot depending on the type of the overlay and the configuration parameters. One important parameter is the failure detection period that affects the delay between a node failure and the failure notification issued by the SON. This delay is configurable and was not counted when evaluating the fault recovery.

Another source of message overhead is the leader election algorithm. Fig. 4(c) shows the average number of leader election messages versus replication degree. The number of messages increases linearly with increasing number of replicas. This overhead is configurable and affects the period between the leader failure and the election of a new leader. In our simulation this period was configured to be maximum 10 seconds. This period is on the critical path and affects the execution of requests as discussed in section V-B1.

VI. Related Work

For the implementation of the RME abstraction we adopt the replicated state machine approach [5] which is routinely used to build stateful fault-tolerant algorithms and systems. For consensus among replicas on the sequence of input events, our implementation deploys the so-called ”Multi-Paxos” [11] version of the Paxos protocol [11], [17] where all proposals from the same leader until its demotion share one single ballot. Other specialized versions of Paxos addressing latency and message complexity (e.g. [18], [19]) can clearly be used instead when appropriate. If input events do not interfere with each other and can be processed in any order yielding the same results and replica state, the Generalized Consensus can be used [20], similarly to relaxing the total order broadcast with generic broadcast [21]. State machine replication can be made tolerant to Byzantine failures [22].

For reconfiguration of the replicate state machine we use the SMART approach [6] which builds on the original idea by Lamport to treat the information about system configuration explicitly as a part of its state [11]. Recently, Lamport also proposed similar extensions to Paxos that enable system reconfiguration by transition through a series of explicit configuration with well-defined policies on proposal numbering [23].

The major alternative way to ensure consistency among replicas is to use a group communication protocol such as Virtual Synchrony [24]. In a Virtual Synchrony system processes (replicas in our case) are organized in groups, and messages sent by group members arrive to all group members in the same order; the system also notifies all group members about joins and leaves of group members. Between membership changes virtual synchrony systems would use a non-uniform total order broadcast, while membership changes requires fault-tolerant consensus. We could deploy our replica group management protocol with state machine replication using a group communication middleware, but from the practical point of view it appeared to be simpler to implement from scratch a version of reconfigurable Paxos.

VII. Conclusions and Future Work

We have proposed the concept of Robust Management Elements (RMEs) which are able to heal themselves under continuous churn. Using RMEs allows the developer to separate the issue of robustness of management from the actual management mechanisms. This will simplify the construction of robust autonomic managers. We have presented an approach to achieve RMEs which uses replicated state machines and relies on our proposed algorithms to automate replicated state machine migration in order to tolerate churn. Our approach uses symmetric replication, which is a replica placement scheme used in structured overlay networks to decide on the placement of replicas and uses SON to monitor them. The replicated state machine is used, besides its main purpose of providing the service, to process monitoring information and to decide when to migrate. Although in this paper we discussed the use of our approach to achieve RMEs, we believe that this approach is generic and can be used to replicate other services.

In order to validate and evaluate our approach, we have developed a prototype and conducted various simulation experiments which have shown the validity and feasibility of our approach. Evaluation has shown that the performance (latency and number of messages) of our approach mostly depends on the replication degree rather than on the overlay size.

In our future work, we will evaluate our approach on larger scales and extreme values of load and churn rate. We will optimise the algorithms in order to reduce the amount of messages and improve performance. We intend to implement our approach in the Niche platform to support RMEs in self-managing distributed applications. Finally, we will try to apply our approach to other problems in distributed computing.

References

[1] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," Oct. 15 2001.

[2] IBM, "An architectural blueprint for autonomic computing, 4th edition," http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf, June 2006.

[3] A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, "Enabling self-management of component based distributed applications," in *From Grids to Service and Pervasive Computing*, T. Priol and M. Vanneschi, Eds. Springer, 2008, pp. 163–174.

[4] Niche homepage. [Online]. Available: http://niche.sics.se/

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[6] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *EuroSys'06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. ACM, 2006, pp. 103–115.

[7] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *Communications Surveys & Tutorials, IEEE*, vol. 7, no. 2, pp. 72–93, 2005.

[8] A. Ghodsi, "Distributed k-ary system: Algorithms for distributed hash tables," Ph.D. dissertation, Royal Institute of Technology (KTH), 2006.

[9] J. S. Kong, J. S. Bridgewater, and V. P. Roychowdhury, "Resilience of structured P2P systems under churn: The reachable component method," *Computer Communications*, vol. 31, no. 10, pp. 2109–2123, June 2008.

[10] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[11] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[12] D. Malkhi, F. Oprea, and L. Zhou, "*Omega* meets Paxos: Leader election and stability without eventual timely links," in *Distributed Computing, 19th International Conference (DISC' 05*, ser. LNCS, P. Fraigniaud, Ed., vol. 3724. Cracow, Poland: Springer, Sep. 26–29 2005, pp. 199–213.

[13] C. Arad, J. Dowling, and S. Haridi, "Building and evaluating P2P systems using the Kompics component framework," in *Peer-to-Peer Computing (P2P'09)*. IEEE, Sep. 2009, pp. 93–94.

[14] "Meridian: A lighweight approach to network positioning," http://www.cs.cornell.edu/People/egs/meridian.

[15] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: estimating latency between arbitrary internet end hosts," in *IMW'02: 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 5–18.

[16] D. Leonard, Z. Yao, V. Rai, and D. Loguinov, "On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks," *IEEE/ACM Trans. Networking*, vol. 15, no. 3, pp. 644–656, Jun. 2007.

[17] B. M. Oki and B. H. Liskov, "Viewstamped replication: a general primary copy," in *PODC'88: 7^{th} Ann. ACM Symp. on Principles of Distributed Computing*. ACM, 1988, pp. 8–17.

[18] L. Lamport and M. Massa, "Cheap Paxos," in *DSN'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. IEEE Computer Society, June 28–July 1 2004, p. 307.

[19] B. Charron-Bost and A. Schiper, "Improving fast Paxos: Being optimistic with no overhead," in *PRDC'06: 12^{th} Pacific Rim International Symp. on Dependable Computing*. IEEE, 2006, pp. 287–295.

[20] L. Lamport, "Generalized consensus and Paxos," MSR, Tech. Rep. MSR-TR-2005-33, Apr. 28 2005.

[21] F. Pedone and A. Schiper, "Generic broadcast," in *Distributed Computing, 13^{th} International Symposium*, ser. LNCS, P. Jayanti, Ed., vol. 1693. Springer, Sep. 27–29 1999, pp. 94–108.

[22] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI'99: 3^{rd} Symp. on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 173–186.

[23] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.

[24] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 123–138, 1987.