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

SICS Technical Report T2010:02
ISSN: 1100-3154
March 4 2010
Swedish Institute of Computer Science
Box 1263, S-164 29 Kista, SWEDEN

***Abstract*—Autonomic managers are the main architectural building blocks for constructing self-management capabilities of computing systems and applications. One of the major challenges in developing self-managing applications is robustness of management elements which form autonomic managers. We believe that transparent handling of the effects of resource churn (joins/leaves/failures) on management should be an essential feature of a platform for self-managing large-scale dynamic distributed applications, because it facilitates the development of robust autonomic managers and hence improves robustness of self-managing applications. This feature can be achieved by providing a robust management element abstraction that hides churn from the programmer.**

In this paper, we present a generic approach to achieve robust services that is based on finite state machine replication with dynamic reconfiguration of replica sets. We contribute a decentralized algorithm that maintains the set of nodes hosting service replicas in the presence of churn. We use this approach to implement robust management elements as robust services that can operate despite of churn. Our proposed decentralized algorithm uses peer-to-peer replica placement schemes to automate replicated state machine migration in order to tolerate churn. Our algorithm exploits lookup and failure detection facilities of a structured overlay network for managing the set of active replicas. Using the proposed approach, we can achieve a long running and highly available service, without human intervention, in the presence of resource churn. In order to validate and evaluate our approach, we have implemented a prototype that includes the proposed algorithm.

***Index Terms*—autonomic computing; distributed systems; self-management; replicated state machines; service migration; peer-to-peer.**

I. Introduction

Autonomic computing [1] is an attractive paradigm to tackle management overhead of complex applications by making them self-managing. Self-management, namely self-configuration, self-optimization, self-healing, and self-protection, is achieved through autonomic managers [2]. An autonomic manager continuously monitors hardware and/or software resources and acts accordingly. Autonomic computing is particularly attractive for large-scale and/or dynamic distributed systems where direct human management might not be feasible.

In our previous work, we have developed a platform called Niche [3], [4] that enables us to build self-managing large-scale distributed systems. Autonomic managers play a major rule in designing self-managing systems [5]. An autonomic manager in Niche consists of a network of management elements (MEs). Each ME is responsible for one or more roles in the construction the Autonomic Manager. These roles are: Monitor, Analyze, Plan, and Execute (the MAPE loop [2]). In Niche, MEs are distributed and interact with each other through events (messages) to form control loops.

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

We propose a Robust Management Element (RME) abstraction that developers can use if they need their MEs to tolerate resource churn. The RME abstraction allows simplifying the development of robust autonomic managers that can tolerate resource churn, and thus self-managing large-scale distributed systems. This way developers of self-managing systems can focus on the functionality of the management without the need to deal with failures. A Robust Management Element should: 1) be replicated to ensure fault-tolerance; 2) survive continuous resource failures by automatically restoring failed replicas on other nodes; 3) maintain its state consistent among replicas; 4) provide its service with minimal disruption in spite of resource join/leave/fail (high availability). 5) be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location). Because we are targeting large-scale distributed environments with no central control, such as peer-to-peer networks, all algorithms should operate in decentralized fashion.

In this paper, we present our approach to achieving RMEs that is based on finite state machine replication with automatic reconfiguration of replica sets. We replicate MEs on a fixed set of nodes using the *replicated state machine* [6], [7] approach. However, replication by itself is not enough to guarantee long running services in the presence of continuous churn. This is because the number of failed nodes (that host ME replicas) will increase with time. Eventually this will cause the service to stop. Therefor, we use *service migration* [8] to enable the reconfiguration 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, based on *Structured Overlay Networks* (SONs) [9], that will use migration to *automatically* reconfigure the set of nodes where the ME replicas are hosted. This will guarantee that the service provided by the RME will tolerate continuous churn. The reconfiguration take place by migrating MEs when needed to new nodes. The major use of SONs in our approach is as follows: first, to maintain location information of the replicas using replica placement schemes such as symmetric replication [10]; second, to detect the failure of replicas and to make a decision to migrate in a decentralized manner; third, to allow clients to locate replicas despite of churn.

The rest of this paper is organised as following: Section II presents the necessary background required to understand the proposed algorithm. In Section III, we describe our proposed decentralized algorithm to automate the migration process. Followed by applying the algorithm to the Niche platform to achieve RMEs in Section IV. Finally, conclusions and future work are discussed in Section V.

II. Background

This section presents the necessary background to understand the approach and algorithm presented in this paper, namely: The Niche platform, Symmetric replication scheme, 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 distributed component programming model, APIs, and a run-time system including deployment service. The main objective of Niche is to enable and to achieve self-management of component-based applications deployed on a dynamic distributed environments 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 run-time environment is a network of distributed containers hosting functional and management components. Niche uses a Chord [9] like structured overlay network (SON) as its communication layer. The SON is self-organising on its own and provides overlay services used by Niche such as name-based communication, distributed hash table (DHT) and a publish/subscribe mechanism for event dissemination. These services are used by Niche to provide 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

We assume the following model of Structured Overlay Networks (SONs) and their APIs. We believe, this model is representative, and in particular it matches the Chord SON. In the model, SON provides the 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 name space. The SON automatically and dynamically divides the responsibility between nodes such that there is always a responsible node for every SON identifier. SON provides a 'lookup' operation that 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. Further more, 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, and nodes are responsible for items with identifiers in the range between the node's identifier and the identifier of the predecessor node. Finally, a SON with a circular identifier space naturally provides for symmetric replication of items on the SON - where replica IDs are placed symmetrically around the identifier space circle.

Symmetric Replication [10] is a scheme used to determine replica placement in SONs. Given an item ID i and a replication degree f, symmetric replication can be used to calculate the IDs of the item's replicas. The ID of the x-th ($1 \leq x \leq f$) replica of the item i is computed according to the following formula:

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

where N is the size of the identifier space.

The IDs of replicas are independent from the nodes present in the system. A *lookup* is used to find the node responsible node for hosting an ID. For the symmetry requirement to always be true, it is required that the replication factor f divides the size of the identifier space N.

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 not considered in this paper. A common way to replicate stateful services is to use the replicated state machine approach [6]. In this approach several nodes (replicas) run the same service in order for service to survive node failures.

Using the replicated state machine 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 (requests or commands) and initial state. This means that the service should avoid sources of nondeterminism such as using local clocks, random numbers, and multi-threading.

Replicated state machines, given a deterministic service, can use the Paxos [7] algorithm to ensure that all services execute the same input in the same order. The Paxos algorithm relies on a leader election algorithm [11] that will elect one of the replicas as the leader. The leader ensures the order of inputs by assigning client requests to slots. Replicas execute input sequentially i.e. a replica can execute input from slot $n + 1$ only if it had already executed input from slot n.

The Paxos algorithm 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. This is because Paxos requires that there will always be a *quorum* of alive replicas in the system. The size of the quorum is $(R/2)+1$, where R is the initial number of replicas in the system. In this paper, we consider only fail-stop model (i.e., a replica will fail only by stopping) and will not consider other models such as Byzantine failures.

D. Migrating Stateful Services

SMART [8] is a technique for changing the set of nodes where a replicated state machine runs, i.e. migrate the service. The fixed set of nodes, where a replicated state machine runs, is called a *configuration*. Adding and/or removing nodes (replicas) in a configuration will result in a new configuration.

SMART is built on the migration technique outlined in [7]. The idea is to have the current configuration as part of the service state. The migration to a new configuration happens by executing a special request that causes the current configuration to change. This request is like any other request that can modify the state when executed. The change does not happen immediately but scheduled to take effect after α slots. This gives the flexibility to pipeline α concurrent requests to improve performance.

The main advantage of SMART over other migration technique is that it allows to replace non-failed nodes. This enables SMART to rely on an automated service (that may use imperfect failure detector) to maintain the configuration by adding new nodes and removing suspected ones.

An important feature of SMART is the use of configuration-specific replicas. The service migrates from `conf1` to `conf2` by creating a new independent set of replicas in `conf2` that run in parallel with replicas in `conf1`. The replicas in `conf1` are kept long enough to ensure that `conf2` is established. This simplify the migration process and help SMART to overcome problems and limitations of other techniques. This approach can possibly result in many replicas from different configurations to run on the same node. To improve performance, SMART uses a shared execution module that holds the state and is shared among replicas on the same node. The execution module is responsible for modifying the state by executing assigned requests sequentially and producing output. Other that that each configuration runs its own instance of the Paxos algorithm independently without any sharing. This makes it, from the point of view of the replicated state machine instance, look like as if the Paxos algorithm is running on a static configuration.

Conflicts between configurations are avoided by assigning a non-overlapping range of slots [FirstSlot, LastSlot] to each configuration. The FirstSlot for `conf1` is set to 1. When a configuration change request appears at slot n this will result in setting LastSlot of current configuration to $n + \alpha - 1$ and setting the FirstSlot of the next configuration to $n + \alpha$.

Before a new replica in a new configuration can start working it must acquire a state from another replica that is at least FirstSlot-1. This can be achieved by copying the state from a replica from the previous configuration that has executed LastSlot or from a replica from the current configuration. The replicas from the previous configuration are kept until a majority of the new configuration have initialised their state.

III. Automatic Reconfiguration of Replica Sets

In this section we present our approach and associated algorithm to achieve robust services. Our algorithm automates the process of selecting a replica set (configuration) and the decision of migrating to a new configuration in order to tolerate resource churn. This approach, our algorithm together with the replicated state machine technique and migration support, will provide a robust service that can tolerate continuous resource churn and run for long period of time without the need of human intervention.

Our approach was mainly designed to provide Robust Management Elements (RMEs) abstraction that is used to achieve robust self-management. An example is our platform Niche [3], [4] where this technique is applied directly and RMEs are used to build robust autonomic managers. However, we believe that our approach is generic enough and can be used to achieve other robust services. In particular, we believe that our approach is suitable for structured P2P applications that require highly available robust services.

Replicated (finite) state machines (RSM) are identified by a constant SON ID, which we denote as RSMID in the following. RSMIDs permanently identify RSMs regardless of node churn that causes reconfiguration of sets of replicas in RSMs. Clients that send requests to RSM need to know only its RSMID and replication degree. With this information clients can calculate identities of individual replicas according to the symmetric replication scheme, and lookup the nodes currently responsible for the replicas. Most of the nodes found in this way will indeed host up-to-date RSM replicas - but not necessarily all of them because of lookup inconsistency and node churn.

Failure-tolerant consensus algorithms like Paxos require a fixed set of known replicas we call configuration in the following. 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. First, 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 and all replica addresses (configuration), thus newly created replicas perceive each other as a group and can communicate with each other. RSMID is also distributed to future RSM clients. Whenever clients need to contact a RSM, they resolve the RSMID similar to the initiator and multicast their requests to obtained addresses.

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 runs on all nodes of the overlay and uses several sources of events and information, including SON node failure notifications, SON notifications about change of responsibility, and messages from clients. We discuss the algorithm in greater detail in the following.

Our algorithm is built on top of Structured Overlay Networks (SONs) because of their self-organising features and resilience under churn [12]. The algorithm exploits lookup and failure detection facilities of SONs for managing the set of active replicas. Replica placement schemes such as symmetric replication [10] is used to maintain location information of the replicas. This is used by the algorithm to select replicas in the replica set and is used by the clients to determine replica locations in order to use the service. Join, leave, and failure events are used to make a decision to migrate in a decentralized manner. Other useful operations, that can be efficiently built on top of SONs, include multi-cast and range-cast. We use these operations to recover from replica failures.

A. 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 replica set). The architecture of a replica (a state machine) that supports migration is shown in Fig. 1. The architecture uses the shared execution module optimization presented in [8]. This optimization is useful when the same replica participate in multiple configurations. The execution module captures the logic of the service. The execution module executes requests. The execution of a request may result in state change, producing output, or both. The execution module should be a deterministic program. 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. That is the state can be externally saved and restored. This enables us to transfer states between replicas. The execution module executes all requests except the `ConfChange` request which is handled by the state machine.

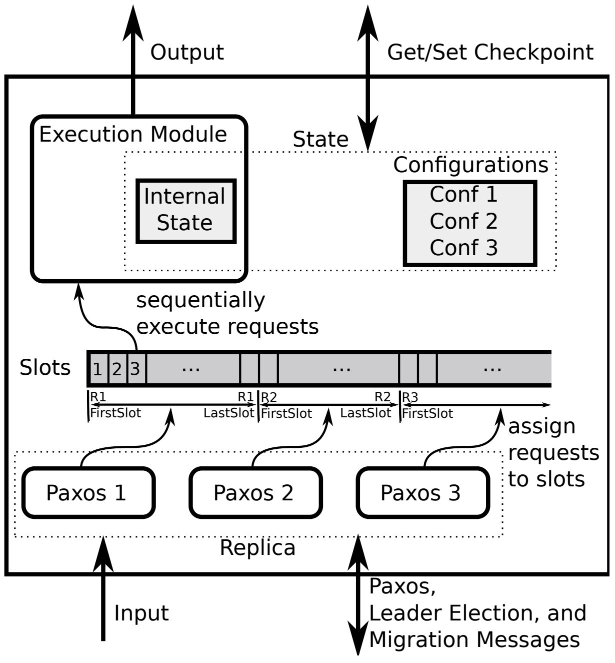

Fig. 1. State Machine Architecture: Each machine can participate in more than one configuration. A new replica instance is assigned to each configuration. Each configuration is responsible for assigning requests to a none overlapping range of slot. The execution module executes requests sequentially that can change the state and/or produce output.

The state of a replica consists of two parts: The first part is internal state of the execution module which is application specific; The second part is the configuration. A configuration is represented by an array of size f where f is the replication degree. The array holds direct *references* (IP and port) to the nodes that form the configuration. The reason to split the state in two parts, instead of keeping the configuration in the execution module, is to make the development of the execution module 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.

The remaining parts of the SM, 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,

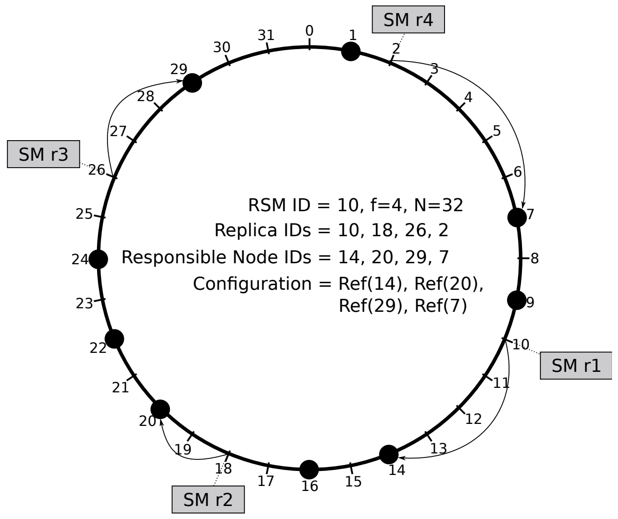
SM r4
SM r3
SM r1
SM r2
RSM ID = 10, f=4, N=32
Replica IDs = 10, 18, 26, 2
Responsible Node IDs = 14, 20, 29, 7
Configuration = Ref(14), Ref(20), Ref(29), Ref(7)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

Fig. 2. 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.

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 when executing `ConfChange` requests, and destroying old configurations after a new configuration is established.

Algorithm 1 Helper Procedures

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

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

B. Configurations and Replica Placement Schemes

All nodes in the system are part of a structured overlay network (SON) as shown in Fig. 2. The Replicated State Machine that represents the service is assigned a random ID $RSMID$ from the identifier space N. The set of nodes that will form a configuration are selected using the symmetric replication technique [10]. 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 and port) of the responsible nodes. These operations are shown in Algorithm 1.

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 replicatino 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. 2, when a node with $ID = 5$ joins the overlay it becomes responsible for the replica `SM_r4` with $ID = 2$. 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 constraints, 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 SM may change. For example in Fig.2 if node 20 fails then node 22 will become responsible for identifier 18 and should host `SM_r2`. Our algorithm, described in the remainder of this section, will automate migration process by triggering `ConfChange` requests when churn changes responsibilities. This will guarantee that the service provided by the RSM will tolerate churn.

C. Replicated State Machine Maintenance

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

1) State Machine Creation: A new RSM can be created by any node in the SON by calling `CreateRSM` shown in Algorithm 2. The creating node construct 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 an `Init SM` message (Algorithm 5) will start a state machine (SM) regardless of its responsibility. Note that the initial configuration, due to lookup inconsistency, may contain some incorrect nodes. This

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
```

does not cause problems for the replication algorithm. Using migration, the configuration will eventually be corrected.

2) Client Interactions: A client can be any node in the system that requires the service provided by the RSM. The client need only to know the $RSMID$ to be able to send requests to the service. Knowing the $RSMID$, the client can calculate the current configuration using equation 1 and lookup operations (See Algorithm 1). 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` as shown in Algorithm 2. The method simply 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. We assume that the client's view overlaps, at least at one node, with the actual configuration for the client to be able to submit requests. Otherwise, the request will fail and the client need to try again later after the system heals itself. We also 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 for efficiency. 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.

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 13) when it finds out that it is not responsible for the target RSM.

On the other hand, it is possible that the configuration was created with some incorrect references. In this case, the node that discovers that it was supposed to be in the configuration will attempt to correct the configuration by replacing the incorrect reference with the reference to itself (Algorithm 3 line 11).

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
4:         if $SM.leader = n$ then
5:             $SM.submit(Req)$
6:         else
7:             sendto $SM.leader$ :
               SUBMIT($RSMID, rank, Req$)
                                  ▷ forward the request to the leader
8:         end if
9:     else
10:        if $r(RSMID, rank) \in ]n.predecessor, n]$ then
                                                  ▷ I'm responsible
11:            JOINRSM($RSMID, rank$)
12:        else
13:            DONOTHING
                      ▷ This is probably due to lookup inconsistency
14:        end if
15:    end if
16: end receipt

17: procedure EXECUTESLOT(req)
                      ▷ This is called when executing the current slot
18:     if req.type = ConfChange then
19:         $newConf \leftarrow Conf[CurrentConf]$
20:         $newConf[req.rank] \leftarrow req.id$
            ▷ Replaces the previous responsible with the new one
21:         $SM.migrate(newConf)$
          ▷ SMART will set LastSlot and start new configuration
22:     else
                      ▷ Execution module handles all other requests
23:         $ExecutionModule.Execute(req)$
24:     end if
25: end procedure
```

3) Request Execution: The execution is initiated by receiving a submit request from a client. This will result in scheduling the request for execution by assigning it to a slot that is agreed upon among all SMs in the configuration (using the Paxos algorithm). Meanwhile, scheduled requests are executed sequentially in the order of their slot numbers. These steps are shown in Algorithm 3.

When a node receives a request from a client it will first check if it is hosting an SM, which the request is directed to. If this is the case, then the node will try to schedule the request if the node believes that it is the leader. Otherwise the node will forward the request to the leader. On the other hand, if the node is not hosting an SM with the RSMID in the request, it will proceed as described in section III-C2, i.e. it ignores the request, if it is not resposible for the target SM, otherwise, it tries to correct the configuration

At execution time, the execution module will execute all requests sequentially except the `ConfChange` request that is handled by the SM. The `ConfChange` request will start the migration protocol presented in [8] and outlined in Section II-D.

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

4) Handling Churn: Algorithm 4 shows how to maintain the replicated state machine in case of node join/leave/failure. When any of these cases happen, a new node may become responsible for hosting a replica. In case of node join, the new node will send a message to its successor to get any replica that now it is responsible for. In case of leave, the leaving node will send a message to its successor containing all replicas that it was hosting. In the case of failure, the successor of the failed node need to discover if the failed node was hosting any SMs. This is done by checking all intervals (line 7) that are symmetric to the interval that the failed node was responsible for. One way to achieve this is by using interval-cast that can be efficiently implemented on SONs e.g. using bulk operations [10].

All newly discovered replicas are handled by `NewSMs` (Algorithm 5). The node will request a configuration change by joining the corresponding RSM for each new SM. 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

In order to validate and evaluate the proposed approach to achieve robust services, we have implemented our proposed

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($newSMs$)
           ▷ SMs are destroyed later by migration protocol
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
```

algorithm together with the replicated state machine technique and migration support using the Kompics component framework [13]. We intend to integrate the implemented prototype with the Niche platform and use it for building robust management elements for self-managing distributed applications. We have conducted a number of tests to validate the algorithm. We are currently conducting simulation tests, using Kompics simulation facilities, to evaluate the performance of our approach.

The autonomic manager in Niche is constructed from a set on management elements. To achieve robustness and high availability of Autonomic Managers, in spite of churn, we will apply the algorithm described in the previous section to management elements. Replicating management elements and automatically maintaining them will result in what we call Robust Management Element (RME). An RME will:

- be replicated to ensure fault-tolerance. This is achieved by replicating the service using the replicated state machine algorithm.
- survive continuous resource failures by automatically restoring failed replicas on other nodes. This is achieved using our proposed algorithm that will automatically migrate the RME replicas to new nods when needed.
- maintain its state consistent among replicas. This is guaranteed by the replicated state machine algorithm and the migration mechanism used.

- provide its service with minimal disruption in spite of resource join/leave/fail (high availability). This is due to replication. In case of churn, remaining replicas can still provide the service.
- be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location). The clients need only to know the `RME_ID` to be able to use an RME regardless of the location of individual replicas.

The RMEs are implemented by wrapping ordinary MEs inside a state machine. The ME will serve as the execution module shown in Fig. 1. 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.

Typically, in replicated state machine approach, a client sends a request that is executed by the replicated state machine and gets a result back. In our case, to implement feedback loops, we have two kinds of clients from the point of view of an RMS. A set of sending client C_s that submit requests to the RME and a set of receiving clients C_r that receive results from the RME. The C_s includes sensors and/or other (R)MEs and the C_r includes actuators and/or other (R)MEs.

To simplify the creation of control loops that are formed in Niche by connecting RMEs together, we use a publish/subscribe mechanism. The publish/subscribe system delivers requests/responses to link different stages (RMEs) together to form a control loop.

V. Conclusions and Future Work

In this paper, we presented an approach to achieve robust management elements that will simplify the construction of autonomic managers. The approach uses replicated state machines and relies on our proposed algorithm to automate replicated state machine migration in order to tolerate churn. The algorithm uses symmetric replication, which is a replication scheme used in structured overlay networks, to decide on the placement of replicas and to detect when to migrate. Although in this paper we discussed the use of our approach to achieve robust management elements, we believe that this approach might be used to replicate other services in structured overlay networks in general.

In order to validate and evaluate our approach, we have implemented a prototype that includes the proposed algorithms. We are currently conducting simulation tests to evaluate the performance of our approach. In our future work, we will integrate the implemented prototype with the Niche platform to support robust management elements in self-managing distributed applications. We also intend to optimise the algorithm in order to reduce the amount of messages and we will investigate the effect of the publish/subscribe system used to construct control loops and try to optimise it. Finally, we will try to apply our approach to other problems in the field of 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 US, July 2008, pp. 163–174.

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

[5] A. Al-Shishtawy, V. Vlassov, P. Brand, and S. Haridi, “A design methodology for self-management in distributed environments,” in *Computational Science and Engineering, 2009. CSE ’09. IEEE International Conference on*, vol. 1. Vancouver, BC, Canada: IEEE Computer Society, August 2009, pp. 430–436. [Online]. Available: http://dx.doi.org/10.1109/CSE.2009.301

[6] 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.

[7] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 51–58, December 2001. [Online]. Available: http://dx.doi.org/10.1145/568425.568433

[8] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, “The SMART way to migrate replicated stateful services,” *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 103–115, 2006.

[9] 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, Second Quarter 2005.

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

[11] D. Malkhi, F. Oprea, and L. Zhou, “Omega meets paxos: Leader election and stability without eventual timely links,” in *Proc. of the 19th Int. Symp. on Distributed Computing (DISC’05)*. Springer-Verlag, Jul. 2005, pp. 199–213. [Online]. Available: ftp://ftp.research.microsoft.com/pub/tr/TR-2005-93.pdf

[12] 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. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2008.01.051

[13] C. Arad, J. Dowling, and S. Haridi, “Building and evaluating p2p systems using the kompics component framework,” in *Peer-to-Peer Computing, 2009. P2P ’09. IEEE Ninth International Conference on*, sept. 2009, pp. 93 –94.