Highly Available Primary-Backup Mechanism for Internet Services with Optimistic Consensus

Koji Hasebe, Naofumi Nishita, and Kazuhiko Kato
Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba 305-8573, Japan
hasebe@cs.tsukuba.ac.jp, nishita@osss.cs.tsukuba.ac.jp, kato@cs.tsukuba.ac.jp

Abstract—**We present an optimistic primary-backup (so-called passive replication) mechanism for highly available Internet services on intercloud platforms. Our proposed method aims at providing Internet services despite the occurrence of a large-scale disaster. To this end, each service in our method creates replicas in different data centers and coordinates them with an optimistic consensus algorithm instead of a majority-based consensus algorithm such as Paxos. Although our method allows temporary inconsistencies among replicas, it eventually converges on the desired state without an interruption in services. In particular, the method tolerates simultaneous failure of the majority of nodes and a partitioning of the network. Moreover, through interservice communications, members of the service groups are autonomously reorganized according to the type of failure and changes in system load. This enables both load balancing and power savings, as well as provisioning for the next disaster. We demonstrate the service availability provided by our approach for simulated failure patterns and its adaptation to changes in workload for load balancing and power savings by experiments with a prototype implementation.**

I. Introduction

With the growth of cloud computing, Internet services merged onto a common platform have become popular. In recent years, the interoperability of clouds has been thoroughly investigated to achieve higher levels of service that can cope with serious damage to a data center, provide computing power for the real-time processing of large amounts of data, and realize elastic management of huge computing resources.

In terms of the availability and fault tolerance of Internet services, replication is a widely used approach. Despite the simplicity of this technique, its implementation is complex because replicating the service state on multiple physical nodes requires that each replica remain synchronized and consistent with the others. To realize consistency in replication, the well-known primary-backup replication method (i.e., passive replication) is often used [3]. In this model, one of the nodes (the primary) processes requests from clients and provides responses. The other nodes (backups) periodically receive state-update messages from the primary, allowing them to update their state to match that of the primary. If the primary fails, one of the backups is selected to take over as the new primary. In implementing this technique, a consensus algorithm (see [4]) is often used to reach agreement among all nodes regarding which is the current primary and which update message is the latest. A well-known consensus algorithm is Paxos [14]. Generally speaking, this algorithm guarantees that $2F + 1$ nodes will achieve consensus in an environment where up to F nodes may simultaneously fail. However, if the majority of nodes fail, the algorithm terminates without achieving consensus. Thus, an implementation of primary-backup replication with such a majority-based algorithm cannot continue to provide services if more than half of the nodes fail, for example, as the result of a large-scale disaster. This limitation is, as Brewer's CAP theorem [1], [10] suggests, caused by the impossibility of achieving both availability and consistency for services.

The objective of this research is to investigate a mechanism to ensure high availability for Internet services based on the primary-backup replication technique. In particular, the Internet service platform based on our proposed method aims to provide services despite a simultaneous failure of the majority of nodes and a partitioning of the network. To achieve this, we take an alternative optimistic approach; instead of using a majority-based consensus algorithm, we use a modified Paxos algorithm that allows agreement to be reached by fewer than half of the nodes. Although this algorithm temporarily permits an inconsistent situation (i.e., nodes may agree on different states), it eventually reaches consistency even if a simultaneous failure of the majority of nodes or a division of the network occurs.

With this algorithm, the Internet services provided on the system avoid interruption by a large-scale disaster. More precisely, a system based on our mechanism continues to provide services as long as at least one node survives. Nevertheless, owing to the modification of Paxos, our system may demonstrate illegal behavior at certain times. Typically, when a network partition divides into node groups consisting of a primary and backups, one node becomes the primary in each separate subgroup, and there may be multiple primaries for a service. Thus, although our method cannot be applied to a service that is required to maintain strict consistency, it is quite useful for Internet services that require high availability but not strict consistency, such as message boards for communication during natural disasters.

Moreover, we incorporate a mechanism for load balancing and the reduction of power consumption in the system. In recent years, there have been a number of suggestions for achieving these goals in cloud data centers (see [2]). One key idea commonly seen in the literature is to diffuse and skew the system's workload according to its changes over time. Based on this approach, in our method the members of a service group are autonomously reorganized over nodes by migrating stored data. This happens in accordance with changes in the system load.

In this paper, we also demonstrate availability through some failure patterns and evaluate the effectiveness of the load-balancing and power-saving mechanisms through experiments with a prototype implementation.

The rest of the paper is organized as follows: Section II discusses related work. Section III provides an overview of Paxos and explains how we modify the algorithm. Section IV presents a system design using our method as a typical application in an intercloud environment. Section V describes our proposed primary-backup mechanism. Section VI demonstrates service availability given various patterns of failure through simulations. Finally, Section VII concludes and recommends future work.

II. Related Work

Various techniques have been proposed to realize replication mechanisms, and these can be classified into two main categories: state machine and primary-backup replication.

In state machine (i.e., active) replication [17], each node processes requests from the clients and transitions independently. Generally, each transition is coordinated across the nodes by means of a consensus algorithm. Although this technique is useful, as a result of its low response time, it has two significant drawbacks: there is a high computational cost, and client requests must be processed in a deterministic manner. On the other hand, primary-backup replication is useful because of its low computational cost and applicability to nondeterministic services. However, as mentioned in Section I, a mechanism is needed that allows agreement on the current primary and its implementation; this is not necessary in state machine replication systems. As explained above, the merits and demerits of these techniques are complementary. To counter these drawbacks, some variant schemes have been proposed in recent years. These include semiactive replication [18] and semipassive replication [6].

However, it is assumed that these techniques will employ a majority-based consensus. Thus, although they guarantee replica consistency, they cannot tolerate the simultaneous failure of a majority of the nodes or partitioning of the network. To address the issue of availability, Dolev et al. [9] proposed optimistic state machine replication based on a self-stabilizing consensus algorithm [7], [8]. Subsequently, Hasebe et al. [13] suggested a self-stabilizing primary backup replication technique and its application to Internet service platforms. The main motivation of our present study is to develop a method for optimistic primary-backup replication using a modified Paxos algorithm. We aim to apply this to Internet service platforms in which the replicas of each service are placed in different data centers.

III. The Paxos Algorithm and Modification

In this section, we first provide an overview of the Paxos algorithm [14] and then explain how to modify it to achieve consensus even if fewer than half of the nodes survive.

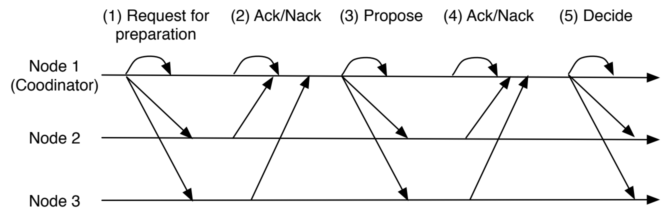

Fig. 1. Actions in each round in Paxos

A. Overview

Consensus is a fundamental problem in distributed computing. It is usually defined as the problem of achieving a goal in light of failures where nonfailed nodes, which started with different inputs, decide on a common output value.

Paxos [14] is a well-known algorithm used to solve this problem. It proceeds in *rounds* and uses a rotating coodinator, i.e., a unique round number i is initially assigned to each node $n_i \in N = \{n_1, \ldots, n_k\}$ and then a node becomes the coodinator based on its current round number. (A possible way to determine the current coordinator is to choose the noncrashed node with the smallest ID.) In each round, the coodinator proposes a preferred value v as the candidate for the output; if this is accepted by the majority of the nodes, the preference is chosen and the algorithm terminates. Otherwise, the coodinator increases its round number by N and switches roles with another node. We note that the round number of each node is uniquely determined at any time, but there may also be multiple coodinators in the system. More precisely, the actions in each round consist of the following five steps (see Fig. 1 for illustration). For readability, we use p_c and p_a to denote the coodinator and the other nodes, respectively.

Step 1. p_c sends a request for preparation of its round number r, denoted by $recPrep(r)$, to all the other nodes.

Step 2. If p_a receives $recPrep(r)$ and p_a has sent Ack to request $recPrep(r')$ or to request acceptance of $recAccept(r')$ with $r' > r$, then it replies $Nack$ to the request. Otherwise, it replies with the message $Ack(r'', v_{\text{prop}}(p_a))$, where $v_{\text{prop}}(p_a)$ is p_a's proposed value and r'' is the round number in $recAccept(r'')$ that was most recently accepted by p_a. Here, if p_a was never accepted, v_{prop} and r'' are set to $Null$ and 0, respectively.

Step 3. p_c waits to receive messages from at least half of the nodes (including itself). If the received messages include $Nack$, then p_c terminates the round and increments r by N. If all the received messages are Ack, then p_c sends the message $recAccept(r, v_{\text{prop}}(p_c))$ to the other nodes; here r is the round number of the current p_c, and $v_{\text{prop}}(p_c)$ is p_c's proposal.

Step 4. If p_a receives $recAccept(r)$ and it has sent Ack to request $recPrep(r')$ or to request acceptance of $recAccept(r')$ with $r' > r$, then it replies $Nack$ to the request. Otherwise, p_a replies Ack to the request.

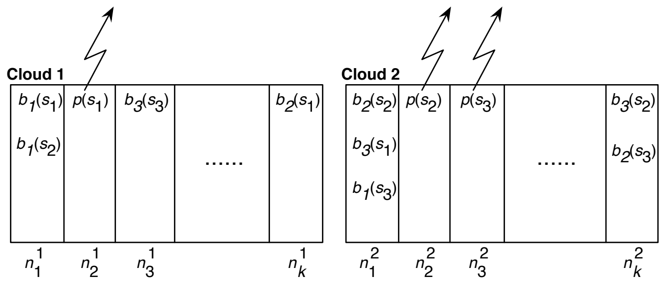

Fig. 2. Overview of the system architecture

Step 5. p_c waits to receive messages from at least half of the nodes (including itself). If the received messages include $Nack$, then p_c terminates the round and increments r by N. Otherwise, p_c decides on $v_{\text{prop}}(p_c)$ as the agreed value and terminates the procedure.

B. Modification of Paxos

As shown by [14], Paxos satisfies the following properties: agreement (i.e., every noncrashed process must agree on the same value), validity (the decided value is one of the inputs), and termination (every noncrashed node eventually decides on a value for output). On the other hand, because steps 3 and 5 require waiting for replies from at least half of the nodes, if a majority of them fail, the coodinator cannot proceed to the next step and the algorithm does not terminate. This limitation is unavoidable whenever the agreement property holds, because if the coodinator node sends its proposed value before receiving messages from half of the nodes, there may be multiple node groups deciding on different values. In other words, guaranteeing agreement (i.e., consistency of outputs) and tolerating a majority of node failures is a trade-off that is inherited from the development of the primary-backup mechanism.

To increase the availability of a primary-backup mechanism, our approach weakens the restrictions at steps 3 and 5, i.e., we modify the algorithm so that the coodinator can propose or decide even if the number of received messages does not reach a majority. (However, to avoid frequent occurrences of inconsistency among replicas caused by a temporary network delay, setting a specific number of required messages that should be received by the coodinator is worthwhile.) Below, we investigate a primary-backup mechanism based on this modified algorithm and its applicability to Internet service platforms.

IV. System Design

Our proposed primary-backup mechanism is intended to be applied primarily to Internet service platforms that are deployed in data centers and clouds. Fig. 2 presents an overview of the architecture of our target platform. Although this platform is developed using a virtualization technique, this discussion is omitted to focus attention on the mechanism itself.

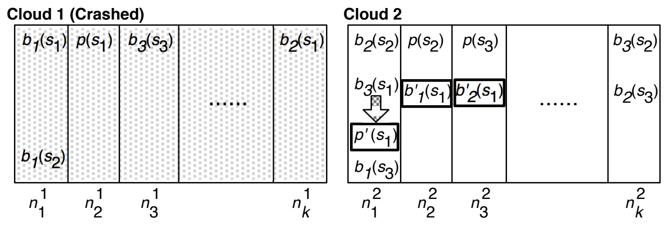

Fig. 3. System behavior after an entire data center fails

To explain the central idea, we consider a simple example that consists of two data centers, "cloud 1" and "cloud 2." The nodes in the two clouds are denoted by $\{n^i_1, n^i_2, \ldots, n^i_{k_i}\}$ with $i = 1, 2$, and multiple servers coexist and provide Internet services. Each service forms a group consisting of a primary and some backups. (It is also possible that several nodes could cooperate as a single primary.) In Fig. 2, we only illustrate three groups for readability. Service s_1 is provided by primary $p(s_1)$ placed on node n^1_2 in cloud 1, while $p(s_2)$ and $p(s_3)$, for services s_2 and s_3, respectively, are placed on n^2_2 and n^2_3 in cloud 2. For each service, the required number of backups and the conditions of their locations can be predetermined. In this case, we consider the requirement that every primary have at least three backups, one of which is placed in a different cloud. For example, $p(s_1)$ has three backups, i.e., $b_1(s_1)$ and $b_2(s_1)$ placed on different nodes in cloud 1 and $b_3(s_1)$ in cloud 2.

When developing a new primary on a node, our mechanism automatically invites nodes that can play a role in its backup. In addition, if a node detects the failure of another node in the same group, the remaining nodes tend to invite a new node to take over for the missing backup and maintain the requirement. In each group, the primary periodically sends its current service state to the backups, and if it fails, one of the backups becomes the new primary and continues to provide service. Owing to our consensus algorithm, services remain as long as at least one node survives.

Fig. 3 illustrates the system behavior when a large-scale disaster happens, i.e., all the nodes in cloud 1 have crashed. In this case, the primary of service s_1 is lost, and thus the only surviving node, $b_3(s_1)$ on n^2_1, becomes the new primary [denoted by $p'(s_1)$, enclosed by a bold rectangle in Fig. 3]. In addition, $p'(s_1)$ asks n^2_2 and n^2_3 to take over the roles of $b'^1_1(s_1)$ and $b'^1_2(s_1)$, respectively. Furthermore, this new node group tends to invite another node from a different cloud to take over the role of $b'^1_3(s_1)$. Similarly, the groups for services s_2 and s_3 complement its backup requirement in a different cloud.

Fig. 4 shows the system's behavior when the backbone network between clouds 1 and 2 fails, making it impossible for nodes in different clouds to communicate. In this case, in each cloud every separated node group considers all nodes in the other cloud to have crashed. Under our mechanism, in cloud 1, $b_1(s_2)$ on n^1_1 and $b_3(s_3)$ on n^1_3 become primaries for s_2 and s_3 and then complement their backups in the same cloud and in different clouds with which it can communicate; in cloud 2,

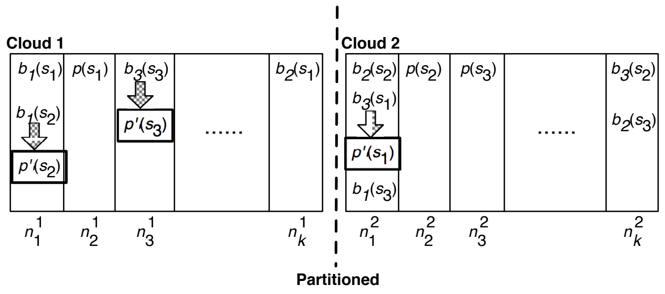

Fig. 4. System behavior after a network partition between clouds

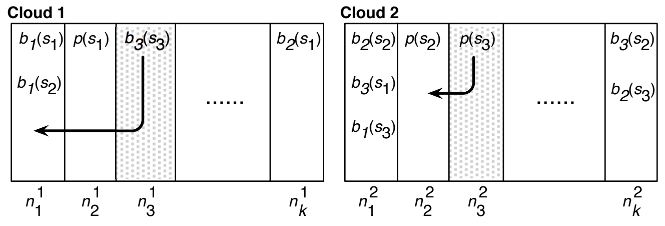

Fig. 5. Off-peak system behavior

$b_3(s_1)$ on n_1^2 becomes the primary for s_1 and complements its backups in a similar way. As this scenario indicates, a network partitioning may lead to the existence of multiple primaries for a service. Thus, each of the primaries provides the same service independently, and the difference between states will increase over time. During such a partition, each node group periodically tries to communicate with the other cloud. If the connection is restored, duplicated primaries merge their states and one of them continues in the primary role while the other disappears. Note that our mechanism allows multiple primaries for a service, whereas in replication based on a majority consensus, all services would stop to strictly avoid any inconsistency. In general, merging different states is a difficult task. However, in the case that a state changes only by the appending of new data, as typified by an Internet message board, the original state can be obtained with relatively little effort.

So far, we have explained the mechanism for fault tolerance. In a final example, we present the mechanism for load balancing and reducing the power consumption of a system. We consider the allocation of primaries and backups as presented in Fig. 2 and assume that the system workload is at peak-time levels. If the load decreases and node n_1^1 can take over the role of $b_3(s_3)$ on n_3^1 without being overloaded, our mechanism automatically moves the state stored by $b_3(s_3)$ from n_3^1 to n_1^1, thereby leaving n_3^1 in a low-power mode (n_3^2 can also be placed into a low-power mode by migrating $p(s_3)$ to n_2^2). When the load increases and a node becomes overtaxed, it relinquishes some of its roles to another node with a lower workload.

Migration of a primary or backup burdens the system network. To reduce the migration cost, retaining the data after a migration for reuse when a primary or backup returns to its original position is worthwhile. This enables migration by copying the difference from the previous migration. The disadvantage of this technique is a trade-off with disk space; however, if the system has enough storage and can afford to create further redundancies, this technique can be useful for effective migration. (See also [12] for details of a similar technique.)

V. Algorithms

Our primary-backup mechanism consists of the following six algorithms:

1) Failure detection
2) Group reorganization
3) Consensus determination
4) Workload management
5) State updating
6) Group unification

Each node periodically executes these algorithms, and these modules act independently and communicate with one another by means of asynchronous message passing in a coordinated manner. The group reorganization algorithm forms a group for a service, and then the members of the group share the roles of primary and backup by the (modified version of) the consensus algorithm. The primary periodically updates the service states using the state-update algorithm to maintain consistency among the backups. When a node fails, the failure detector identifies it and then conveys the information to the group reorganization algorithm. In addition, each node has predetermined upper and lower thresholds, and if the workload exceeds the upper threshold or falls below the lower threshold, the workload manager detects the condition and conveys this information to the group reogranization algorithm. According to the messages from the failure detectors and the workload managers, the group reorganization algorithm periodically executes the consensus algorithm and changes the members of the group. Finally, if the existence of multiple primaries is detected after recovery from a network partitioning, the group unification algorithm merges the separated groups back into the original one.

In the following subsections, we briefly overview these algorithms.

A. *Failure Detection*

Failure detectors are popular applications that are responsible for locating node failures or crashes in a distributed system. Our implementation is based on the usual heartbeat-style technique. That is, each node periodically transmits its own heartbeat packet while simultaneously monitoring the heartbeat packets received from other nodes. If a heartbeat is not observed within a threshold time interval, the receiver considers the sender to have failed.

B. *Group Reorganization*

The group reorganization algorithm manages the forming of a group. This algorithm works in collaboration with the consensus algorithm, the workload manager, and the group unification algorithm. Each group is formed so that it satisfies the following conditions:

1) There exists a single primary node.
2) There exist a backups in the same area.
3) There exist b backups in different areas.
4) All backups are placed in at least c different areas.
5) There is no node whose workload exceeds its upper threshold.

The values of a, b, and c are tunable. If a group cannot satisfy these conditions because of a node or network failure or a change in workload, the consensus algorithm sends the message $lack$ to the group reorganization algorithm. Then the group reorganization algorithm sends $invite$ messages to all the other nodes. A node receiving the $invite$ message replies with an $accept$ message if it can take over for the missing node. Finally, the group reorganization algorithm chooses some of the nodes so as to satisfy these five conditions.

Algorithm 1 Consensus Algorithm

```
while $true$ do
  if $p$ is the coordinator then
    receive messages from the buffer
    if receive $newgroup(group_s)$ then
      $group_s^p \leftarrow group_s$
    end if
    if receive $failure(failedNode)$ then
      $failedNode^p \leftarrow failedNode$
    end if
    if receive $serviceout(p')$ then
      $outNode \leftarrow outNode \cup \{p'\}$
    end if
    while $group_x$ cannot satisfy the group conditions do
      send $lack$ to the group reorganization algorithm
      wait to receive $newgroup(group_s)$ and $group_s^p \leftarrow group_s$
    end while
    $v_{prop} \leftarrow group_s^p$
    execute the procedure for consensus
    $repgroups_s \leftarrow v_{decide}$
    send $repgroups_s$ to the group reorganization algorithm
    wait
  else
    while the coordinator fails do
      receive messages from the buffer
      if receive $newgroup(group_s)$ then
        $group_s^p \leftarrow group_s$
      end if
      if receive $failure(failedNode)$ then
        $failedNode^p \leftarrow failedNode$
      end if
      if receive $serviceout(p')$ then
        $outNode \leftarrow outNode \cup \{p'\}$
      end if
      if receive $recPrep$ or $recAccept$ then
        send $Ack$ or $Nack$
      end if
      if receive $Decide(v)$ then
        $repgroup_s^p \leftarrow v$
        send $repgroups_s$ to the group reorganization algorithm
      end if
    end while
  end if
end while
```

C. Consensus Determination

The consensus algorithm is the heart of our approach. This algorithm is used to reach agreement among group members about which are the primary and the backups, so a decided-upon value determines these assignments.

Our optimistic consensus algorithm behaves as follows: The coodinator can propose or decide if the number of received messages reaches a specific value m, which may represent less than a majority. When a node generates a proposed value, it checks whether a change in the members is needed. If it is, the consensus algorithm sends a request to reorganize the group. If the group reorganization algorithm finds a possible reorganization that satisfies the conditions on group members, the consensus algorithm allocates the roles of the primary and backups to the new members of the group. After the proposal is decided, this decided-upon value is sent to the reorganization algorithm, and then the group is reorganized.

The details are presented in Algorithm 1, where we introduce the following expressions.

- $newgroup(group_s)$: possible members $group_s \subseteq N$ for reorganization of the group of service s
- $failure(failedNode)$: a message that the failed nodes are $failedNode \subseteq N$
- $outNode$ ($\subseteq N$): a list of nodes that need to leave a group
- $serviceout$: a request to leave a group
- $assign\_group_s$: assignment of the primary and the backups.

D. Workload Management

The workload manager monitors the load of its own node. If this exceeds the upper threshold, the manager sends a request to the consensus algorithm to release some of its primaries and backups. In addition, if the workload falls below the upper threshold, the manager sends a request to the consensus algorithm to migrate all primaries and backups to another node. However, if just any request were allowed, frequent migrations could result. Thus, a request is accepted if a primary or a backup migrates from a node with a lower workload to another with a higher load.

E. State Updating

Every primary periodically sends the current state of its service to all backups, while every backup updates its backup state with the primary's message. This procedure is managed by the state-update algorithm.

F. Group Unification

When a network failure divides a group, our optimistic consensus, allows two separate groups to be formed and continue to provide the same service independently. During such a network partitioning, the group unification algorithms periodically send messages to try to communicate with each other, and if the channel is restored, the algorithms in the separated groups send a request to the consensus algorithm to unify them into the original group.

VI. Experiments with Implementation

To verify the correctness of our algorithm and to demonstrate its ability to keep services available, we conducted experiments with our current prototype implementation of the proposed method. In addition, to demonstrate the mechanism

TABLE I
SERVER CAPACITY AND LOWER AND UPPER THRESHOLDS

	Capacity	Upper threshold	Lower threshold
#1–#7	100	90	70
#8–#14	80	70	50
#15–#21	100	90	70
#22–#28	80	70	50

for load balancing and power saving, we used this implementation to simulate a change in the number of active physical nodes and the average load in the course of a day in which the system workload varies.

Our prototype consists mainly of the six modules explained in the Section V. These modules behave independently and communicate with each other by means of asynchronous message passing. To implement these concurrent processes, our prototype was implemented in Scala [16]. In particular, the interprocess communications were implemented by using Scala Actors library. On the other hand, our implementation has not reached a practical level: it does not function as an Internet service platform and cannot execute real application servers.

A. Experimental Setup

The experiments were conducted on 28 PC servers, each of which was equipped with 8 Intel Core i7 2.67 GHz CPUs, 11.8 GB memory, and a single 500 GB ATA disk. Each server was connected to a single switch via a 1000Base-T network adapter.

We virtually introduced two areas in the system; servers 1–14 were placed in area 1, while servers 15–28 were in area 2. As conditions for group membership, each primary must have two backups in the same area and two in the other, unless the required nodes are alive. If a group loses some of its backups in a different area and no node remains to take over the backups in that area, then the group complements them with nodes in the same area. Instead of installing real software, the server workloads were simulated by natural numbers. For each server, the capacity and lower and upper thresholds were set as in Table I.

B. Demonstration of Service Availability

As explained in Section IV, as examples of possible large-scale disasters in a real system, we considered the following two cases:

Case 1. All nodes in area 2 fail.

Case 2. The network between areas 1 and 2 is partitioned.

For each case, there were five services. The primaries for services 1–3 were placed in area 1, while services 4 and 5 were in area 2. Every service has four backups, which satisfies the condition described above. The workloads of the primary and backup were respectively 70 and 20. The results of the experiments are as follows.

Case 1: All nodes in area 2 fail. Fig. 6 illustrates the behavior of the system when a failure occurs after a lapse of 50 s. In this experiment, immediately after the failure,

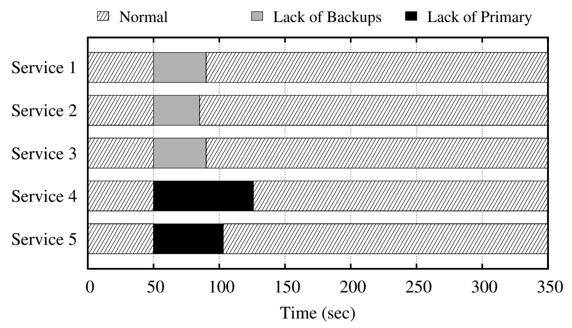

Fig. 6. Case 1: System behavior when all nodes in an area fail

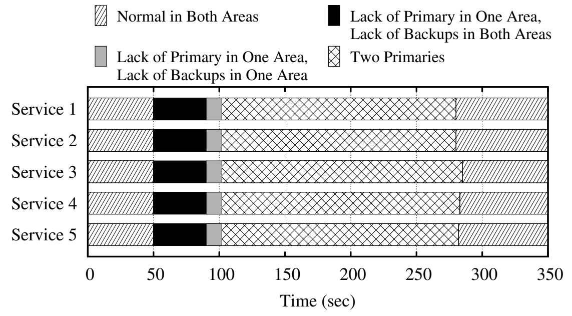

Fig. 7. Case 2: System behavior when network partitioning occurs

services 1–3 lacked backups in area 2, and services 4 and 5 lacked both primaries and some of the backups. After 35–40 s, each of services 1–3 invited two nodes into the group and complemented the backups. On the other hand, after 57–76 s, each of services 4 and 5 also complemented the backups, and one former backup became a primary. We note that for services 4 and 5, the complementing of the backups and the election of a primary were accomplished simultaneously. This is because these procedures are handled by the group organization algorithm.

Case 2: The network is partitioned. Fig. 7 illustrates the behavior of the system when a network failure occurs and how the system recovers, after lapses of 50 and 250 s, respectively. During the failure, none of the nodes in different areas can communicate with each other. In this experiment, immediately after the failure, in area 1 services 1–3 lacked backups and services 4 and 5 lacked both a primary and backups, while in area 2, services 1–3 lacked both a primary and backups and services 4 and 5 lacked backups. After 40 s, services 1–3 in area 1, as well as services 4 and 5 in area 2, obtained the required number of backups. Then, after 12 s, all primaries and backups were complemented. Furthermore, after 250–300 s, for each service the multiple primaries merged into a single one.

From the results of these experiments, we conclude that our mechanism behaves as intended and makes it possible to continue to provide services in environments where more than half the total number of nodes fail or network communication

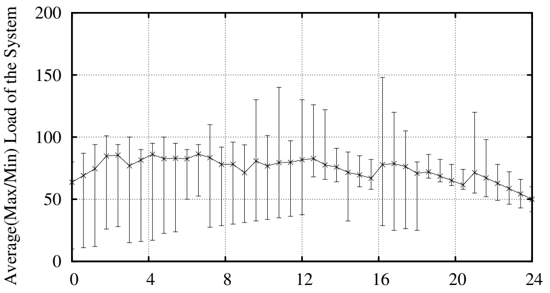

Fig. 8. Average (max/min) system load

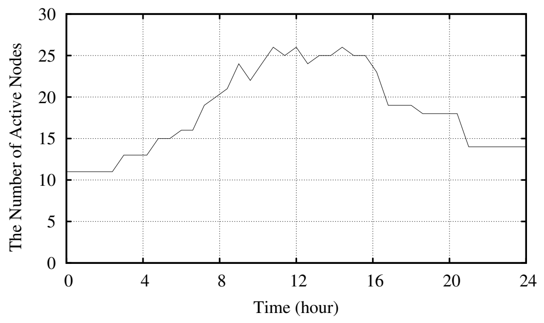

Fig. 9. Number of active nodes

between different areas is lost.

C. Demonstration of Load Balancing and Power Reduction

In this experiment, we introduced 10 services. Five primaries are initially located in each area. In the intended environment, we considered the system load to vary over the day. During the course of a day, the workloads of every primary and backup were initially 30 and 10, respectively, and these were incremented by values of 2 and 1 every 36 minutes until the middle of the day (when the workloads were 70 and 30) and then decreased until the end, respectively. We assumed that a node storing no primary or backups was at low power; otherwise it was active.

Figs. 8 and 9 illustrate the change in the average (as well as maximum and minimum) workload of active nodes and the number of active nodes. Fig. 8 shows that the average load was in the range of 60%–80%, which is about the same as the range of predetermined upper and lower thresholds. However, because of the limitation of allocating backups, our mechanism could not achieve an optimal configuration. Fig. 9 shows that our mechanism adjusts the number of physical nodes to the variation in workload and reduces power consumption effectively. Indeed, if we consider a static configuration (i.e., without migration), it is necessary to activate 20 nodes to deal with the system load in this setting.

VII. Conclusions and Future Work

We have proposed an optimistic primary-backup mechanism for highly available Internet services, the prime application of which is intercloud environments. The key idea is to use an optimistic consensus algorithm that allows agreement to be reached with fewer than half of the of nodes participating, as opposed to a traditional majority-based one. In addition, through interservice communications, members of service groups are autonomously reorganized according to the type of failure and changes in system load. This enables both load balancing and power saving, as well as provisioning for the next failure. Experiments show that our mechanism indeed behaves in the intended manner and can tolerate various failure patterns, including a simultaneous failure of the majority of nodes or a partitioning of the network. Moreover, our mechanism effectively skews workload, thereby reducing the running time of active nodes.

In future work, we will investigate further to refine our prototype implementation. In particular, we intend to implement functions to provide real services on the system for practical use.

References

[1] E. A. Brewer. Towards robust distributed systems (Invited Talk). *Principles of Distributed Computing (PODC 2000)*, 2000.

[2] T. Bostoen and S. Mullender. Power-Reduction Techniques for Data-Center Storage Systems. *ACM Computing Surveys*, vol.45(3), 2013.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed Systems (2nd ed.)*, ACM press/Addison-Wesley Publishing Co., pp.199-216, 1993.

[4] G. Coulouris, J. Dollimore, and T. Kindberg. Chapter 12 in *Distributed Systems: Concepts and Design (4th Edition)*, Addison Wesley, 2005.

[5] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. *USENIX NSDI'08*, pp.161-174, 2008.

[6] X. Defago and A. Schiper. Semi-Passive Replication and Lazy Consensus. *Journal of Parallel and Distributed Systems*, vol.64(12), pp.1380-1398, 2004.

[7] E. W. Dijkstra. Self-Stabilizing System in Spite of Distributed Control. *Communication of the ACM*, vol.17(11), pp.643-644, 1974.

[8] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[9] S. Dolev, R. I. Kat, and E. M. Schiller. When Consensus Meets Self-Stabilization. *Journal of Computer and System Science*, vol.76(8), pp.884-900, 2010.

[10] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, vol.33(2), pp.51-59, 2002.

[11] R. Guerraoui and A. Schiper. Consensus Service: a modular approach for building agreement protocols in distributed system. *Annual Symposium on Fault Tolerant Computing*, pp.168-177, 1996.

[12] K. Hasebe, T. Niwa, A. Sugiki, and K. Kato. Power-Saving in Large-Scale Storage Systems with Data Migration. *2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, pp. 266-273, 2010.

[13] K. Hasebe, K. Yamatozaki, A. Sugiki, and K. Kato. Self-Stabilizing Passive Replication for Internet Service Platforms. *4th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, 6 pages, 2011.

[14] L. Lamport. The Part-time Parliament. *ACM TOCS*, vol.16(2), pp.133-169, 1998.

[15] D. Mazieres. Paxos Made Practical. Technical report, 2007 (available at http://www.scs.stanford.edu/ dm/home/papers/).

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*, Artima, 2008.

[17] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, vol.22(4), pp.299-319, 1990.

[18] D. Stodden. Semi-Active Workload Replication and Live Migration with Paravirtual Machines. *Xen Summit*, 2007.