FRAPPÉ: Fast Replication Platform for Elastic Services

Vita Bortnikov[1], Gregory Chockler[1], Dmitri Perelman[2], Alexey Roytman[1], Shlomit Shachor[1], and Ilya Shnayderman[1]

[1]IBM Research
[2]Technion, Israel Institute of Technology

August 1, 2011

Abstract

Elasticity is critical for today's cloud services, which must be able to quickly adapt to dynamically changing load conditions and resource availability. We introduce FRAPPÉ, a new consistent replication platform aiming at improving elasticity of the replicated services hosted in clouds or large data centers. In the core of FRAPPÉ is a novel replicated state machine protocol, which employs *speculative executions* to ensure continuous operation during the reconfiguration periods as well as in situations where failures prevent the agreement on the next stable configuration from being reached in a timely fashion. We present the FRAPPÉ's architecture and describe the basic techniques underlying the implementation of our speculative state machine protocol.

1 Introduction

Replicated state machine [9] is an important tool for maintaining integrity of distributed applications and services in failure-prone data center and cloud computing environments. In massively multi-tenant settings of today's clouds, large number of replica groups share the common hardware infrastructure, and compete for limited resources. In order to be able to dynamically adapt to changing resource availability, load fluctuations, variable power consumption, and support better data locality, the consistent replication layer must be *elastic*, i.e., being capable of supporting dynamically changing replication groups with a minimum disruption to the service availability and performance.

However, to preserve correctness across configuration changes, the reconfiguration protocol must ensure that the state machine execution responsibilities have been transferred to the members of the new configuration in an orderly fashion, and in particular, no new user commands are executed in the new configuration before it has been agreed by the members of the old one. The resulting throughput degradation might be prohibitive if the rate of dynamic changes is high. Furthermore, the service availability will suffer if the old configuration is lost (e.g., due to a failure) before the agreement on the new one has been completed. For example, Amazon Web Services [1] must guarantee 99.99's availability, which translates to less than 52 minutes of unavailability a year. If a significant portion of the service up-time the normal operation is interrupted to execute the reconfiguration protocol, these availability goals might not be met.

In this paper, we introduce a new replication platform, called FRAPPÉ (*Fast ReplicAtion Platform for Elastic services*), which resolves the inefficiencies, and availability limitations associated with dynamic reconfiguration. In the core of FRAPPÉ is a novel replicated state machine protocol, which employs *speculative executions* to ensure continuous operation during the reconfiguration periods as well as in situations where failures prevent the agreement on the next stable configuration from being reached in a timely fashion.

Internally, our speculative state machine implementation is based on the reconfigurable Paxos approach [10, 11, 12]: i.e., configurations are treated as a part of the replicated state, and are being agreed upon in their own consensus instances. However, in contrast to reconfigurable Paxos, the command ordering can continue to execute normally even if the agreement on the configuration relative to which those commands will be ordered is still in progress. The key observation is that the command ordering can be executed in an *estimated* configuration provided the validity of the speculative decisions can be verified once the next *agreed* configuration becomes available.

To accomplish that, in FRAPPÉ, each replica maintains a *branching* command log (see Figure 3) as opposed to a linear log maintained by the standard replicated state machine implementations. Whenever a replica learns of (or proposes) a new speculative configuration, it creates a new log branch originating in the log location associated with the agreement instance created for that configuration. Each branch then proceeds to execute its own independent sequence of the Paxos agreement instances as explained in Section 4.1. Whenever a replica learns the outcome of the agreement for a slot with one or more speculative branches, it stems all branches originating at that slot except for the one whose configuration is identical to the one that has been agreed (if exists).

Speculatively agreed commands can be applied to the state either immediately (at the risk of the possible future rollback), or when their branch is validated against the agreed configuration. Since in a common case, estimated configurations would coincide with those being eventually decided, and the result of the reconfiguration agreement will be available by the time the first speculative command is agreed upon, reconfiguration will have a little impact on the overall command throughput. Furthermore, since incoming commands can continue to be ordered in an estimated configuration, the system availability will be unaffected even when underlying failures prevent the configuration agreement from completion. The global log consistency is guaranteed to be restored once the real configuration is learnt, and the speculative branches are verified against it.

Reconfiguration can be driven by a variety of high-level resource management systems (such as placement controllers, load balancers, and health-monitoring systems), or initiated through the administrative inputs. In FRAPPÉ, the reconfiguration decisions are mediated through the Configuration Manager component (see Figure 2), which plans and triggers configuration changes, based on the current configuration membership, and pluggable reconfiguration policies.

The rest of the paper is organized as follows. Related work is discussed in Section 2. The FRAPPÉ architecture is described in Section 3. Section 4 discusses our speculative replicated state machine implementation, and Section 5 concludes the paper.

2 Related work

Several approaches to alleviating reconfiguration bottleneck in reconfigurable state machines have been proposed. The original idea by Lamport, described in [10, 11, 12], and implemented in SMART [14], was to delay the effect of the configuration agreed in a specific consensus instance by a fixed number α of successive consensus instances. If the configuration must take effect immediately, the remaining instances can be skipped by passing a special "window closure" decree consisting of α consecutive *noop* instances. Although this approach allows up to α consecutive commands to be executed concurrently, choosing the right value of α is nontrivial. On the one hand, choosing α to be too small may under-utilize the available resources. On the other hand, large values of α may not match the actual service reconfiguration rate resulting in too frequent invocations of the window closure decrees (which must complete synchronously).

Chubby [6] and ZooKeeper [8] expose high-level synchronization primitives (respectively, locks and watches) that can be used to implement a reconfigurable state machine within the client groups. The solutions based on this approach are however, vulnerable to timing failures, and therefore, either restrict their failure model [20], or rely on additional synchronization protocols within the replication groups themselves to maintain consistency [13, 17].

Vertical Paxos [13] removes the configuration agreement overhead from the critical path by delegating it to an auxiliary "configuration master". The reconfiguration involves an extra step of synchronizing with the read quorums of all preceding configurations causing throughput degradation. In addition, the configuration master itself is implemented using the α-based reconfigurable Paxos protocol, and therefore, suffers from the limitations similar to those discussed above.

Dynamic reconfiguration has been extensively studied in the context of virtually synchronous group communication [18, 5, 7, 16], and reconfigurable read/write registers [15, 2]. The reconfiguration protocols described in these papers do not aim to support consistency semantics as strong as those of state machine replication, and therefore do not directly apply in our context. Birman et al. [4] present a replication framework unifying reconfigurable state machine and virtual synchrony in which the normal operation is suspended during the reconfiguration periods.

Optimistic and speculative approaches to mask the coordination latency have been extensively studied in the past in a variety of contexts (such as e.g., group communication [19], and database replication [3]). However, to the best of our knowledge, speculative reconfigurable state machine replication studied in this paper has not yet been addressed in the prior work.

3 System Architecture

The members of each elastic service cluster whose state is managed through FRAPPÉ are organized into *replication groups* (see Figure 1). Each member of the replication group can be either *active* or *idle*. The active members

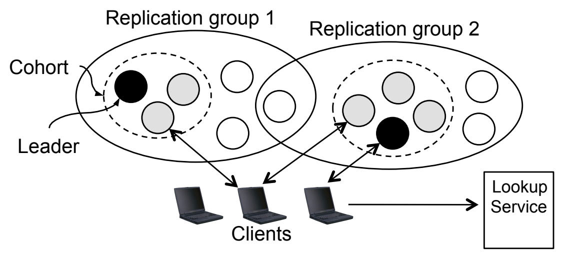

Figure 1: The FRAPPÉ Group Structure

hold the up-to-date copies of the service state, and are responsible for the client and reconfiguration command ordering. The set of all currently active members of a replication group form the group's *cohort*. The ordering protocol is orchestrated by a distinguished member of the cohort, called the *leader*. Although the idle processes do not participate in the command ordering, they are nevertheless available for serving the state transfer requests (see Section 4.2). They can be taken off-line once it is verified their copies of the service state have been propagated to a sufficient number of the group members.

The current cohort configuration and leader identity within each replication group is maintained by the cohort members, and propagated to the idle members through a gossip-based protocol. This information is made available to the clients, and newly joining replicas through an external lookup service, such as DNS or LDAP, which maps each group name to its cohort configuration and leader identity.

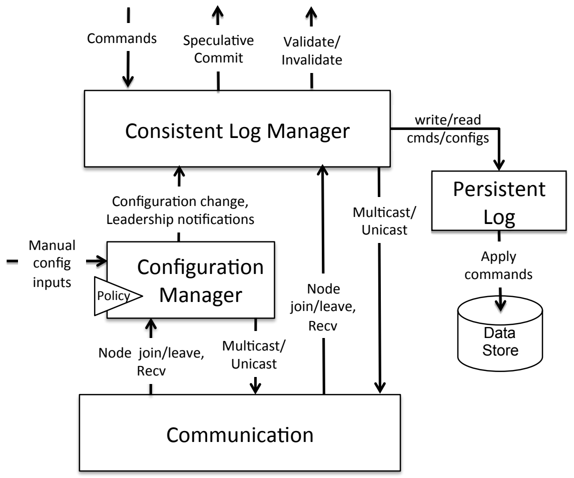

Figure 2: The FRAPPÉ Layers

The layered structure of each individual FRAPPÉ replica is depicted in Figure 2. The user commands are processed by the Consistent Log Manager layer whose responsibility is to maintain globally ordered command log with possible speculative branching as explained in Section 4.1. The command log is persisted on stable storage through the Persistent Log layer. Whenever the size of a locally maintained log grows beyond a configured upper limit, a portion of the globally ordered command prefix is clipped, and the commands in this prefix are applied to the local copy of the replicated state kept on the persistent data store.

Configuration Manager keeps track of the current cohort configuration, and plans and triggers configuration changes, which are passed to Consistent Log Manager for agreement. New cohort members are recruited from the set of the idle members of the replication group. The configuration change decisions are driven by the current view of the replication group, cohort membership, and a pluggable policy. Configuration Manager is also in charge of selecting the cohort leader. Both Consistent Log and Configuration Managers utilize services provided by the Communication layer for disseminating protocol messages, and failure detection.

4 Speculative State Machine

In this section, we discuss the two main parts underlying our speculative replicated state machine implementation, which are speculative command ordering and state transfer protocol, in more details.

4.1 Command Ordering

Our command ordering implementation is based on the standard reconfigurable Paxos approach [10, 11, 12] (also known as "Horizontal Paxos" [13]) augmented with the speculative ordering support. As in standard Paxos, each individual slot in the log is occupied by the user command chosen by an instance of the agreement protocol. The agreement protocol is orchestrated by a single process, called a *proposer*, and consists of two phases each one associated with a unique monotonically increasing ballot number, and consisting of one round-trip message exchange. The first phase chooses the value to propose in the slot, and the second phase stores the chosen value at the replicas. Agreement instances for each slot are independent of each other, and can be executed in parallel. The first phase can be shared across multiple agreement instances provided they are mediated by the same proposer and use the same ballot number. Thus, in the common case when the proposer remains live and connected for a long time, each command is ordered within just two message delays, and multiple command orderings

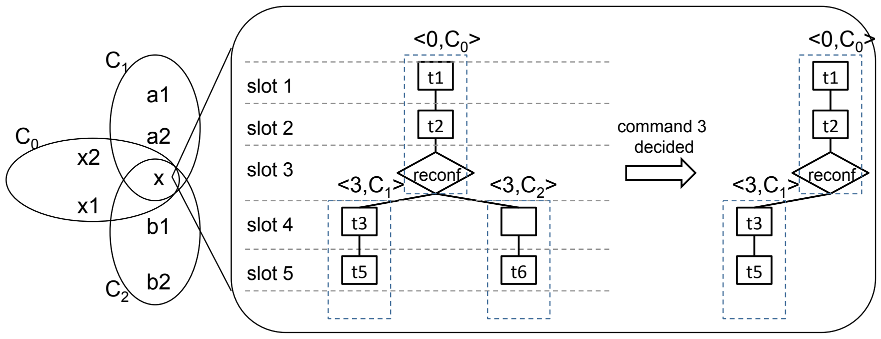

Figure 3: Maintaining multiple branches during reconfiguration. Ballot and branch initiator components of the branch identifiers are omitted for clarity. Node x participates in two speculative branches $\langle 3, C_1 \rangle$ and $\langle 3, C_2 \rangle$. Once x learns the configuration decided by command 3, it prunes branch $\langle 3, C_2 \rangle$ and merges branch $\langle 3, C_1 \rangle$ into the trunk.

are executed in parallel resulting in a high throughput, low latency protocol.

Each command agreement is executed against a quorum of processes of the current replica configuration. In Horizontal Paxos, a configuration is treated as a part of the replicated service state: i.e., the configuration for slot i of the log is determined by the latest configuration change command preceding i in the log. Below, we describe how this basic scheme is extended to allow *speculative* execution of the user commands relative to configuration *estimates* instead of agreed configurations. We will focus on the novel aspects of the speculative state machine implementation, and omit the details of Horizontal Paxos, which are well-known.

The sequence of commands executed relative to a configuration estimate C is represented as a *branch* in the global command log starting at the position allocated for the agreement on C (see Figure 3). Specifically, the branch spawned by a reconfiguration command cmd is uniquely identified by a triple consisting of the following three components (lines Algorithm 1.2–4): (1) slot number occupied by cmd in the parent branch , (2) ballot number that was used to propose cmd, and (3) identifier of the cmd's proposer. The branch identifier is attached to all messages associated with the Paxos agreement instances proposed within the branch, and is used to route received messages to the correct instance of the Paxos protocol (lines Algorithm 2.20–22).

Each branch executes a stream of the Paxos agreement instances, and the agreement streams within different branches proceed independently from each other (see Figure 3). In particular, each branch maintains its own set of the Paxos data structure, a subset of which consisting of the maximum ballot number, the totally ordered command prefix, and the next available agreement slot are exposed to the speculative state machine (lines Algorithm 1.9–11).

Algorithm 1 Types and States for Replica p_i:

```
1:  Record BranchID:
2:      ℕ: $slotNum$        ▷ slot num of the reconfiguration command in the parent branch
3:      Ballot: $bal$        ▷ initial ballot of the branch
4:      PID: $branchInitiator$        ▷ identifier of the reconfiguration command proposer

5:  Record PaxosBranch:
6:      BranchID: $bid$        ▷ identifier of this Paxos branch
7:      BranchID: $parent$        ▷ identifier of the parent branch
8:      PID[]: $config$        ▷ configuration of this Paxos branch
9:      Ballot: $bal$        ▷ maximum ballot number locally known to this Paxos branch
10:     Command[]: $cmdLog$        ▷ prefix of user and reconfiguration commands ordered by this Paxos branch
11:     ℕ: nextSlot        ▷ next available command slot locally known to this Paxos branch

12: Replica State Variables:
13:     Set of PaxosBranch: $branches$        ▷ locally known live Paxos branches
14:     Command[]: $trunk$        ▷ global total order prefix

15: Replica State Initialization:
16:     Let $B_0 = \langle b_0, \perp, C_0, bal_0, \langle\rangle, 1\rangle$ where $C_0$ is the initial configuration, $b_0 = \langle 0, bal_0, q\rangle$, and $q$ is a deterministically chosen member of $C_0$.
17:     $branches = \{B_0\}$
18:     $trunk = \langle\langle \text{RECON}, \perp, b_0, C_0\rangle\rangle$
19:     Start an instance of Paxos protocol for branch $B_0$
```

All the live branches known to a replica are kept in the set $branches$ (line Algorithm 1.13), and are linked together into a tree-like structure through the $parent$ branch identifier associated with each branch (line Algorithm 1.7).

A replica creates a new speculative branch and adds it to its local $branches$ set (line Algorithm 2.32) whenever it receives a JOIN message carrying the new and parent branch identifiers, and the new configuration that spawned the branch. The replica then starts a new instance of the Paxos agreement for that branch initializing it with

Algorithm 2 Message Handlers for Replica p_i:

```
20: Upon receiving $m = \langle \text{USER}, bid, \cdot \rangle$ or $\langle \text{RECON}, bid, \cdot \rangle$:
21:     if ($\exists B \in branches, B.bid = bid$) then
22:         Pass $m$ to the Paxos instance for branch $B$

23: New user command $cmd$:
24:     For each branch $B \in branches$ where $p_i$
        is the proposer for $x$:
25:         Propose $\langle \text{USER}, B.bid, cmd \rangle$ in the instance of Paxos protocol for $B$

26: New reconfiguration command $cmd = \langle C \rangle$:
27:     For each branch $B \in branches$ where $p_i$
        is the proposer for $B$:
28:         Let $x = \langle B.nextSlot, B.bal, p_i \rangle$
29:         Send $\langle \text{JOIN}, B.bid, x, C \rangle$ to all members of $C$
30:         Propose $\langle \text{RECON}, B.bid, x, C \rangle$ in the instance of Paxos protocol for branch $B$

31: Upon receiving: $\langle \text{JOIN}, bid, x, C \rangle$:
32:     Add branch $X = \langle x, bid, C, x.bal, \langle\rangle, x.slotNum + 1 \rangle$
        to $branches$
33:     Start new instance of Paxos protocol for branch $X$, and initiate state transfer (see Section 4.2)
```

Algorithm 3 Total Order Construction at Replica p_i:

```
34: Task Construct Total Order:
35:     $curBranch := B_0$
36:     $next := 1$
37:     while (true) do:
38:         $idx := next - curBranch.bid.slotNumber$
39:         Block until $idx \leq \text{length}(curBranch.cmdLog)$
40:         while ($idx \leq \text{length}(curBranch.cmdLog)$) do:
41:             $cmd := curBranch.cmdLog[idx - 1]$
42:             $B := \{b \in branches : b.parent = curBranch \wedge b.slotNumber = next\}$
43:             if ($cmd = \langle \text{RECON}, \cdot, x, \cdot \rangle \wedge$
                $\exists b \in B : b.bid = x$ then
44:                 Discard $curBranch$, terminate Paxos instance associated with $curBranch$
45:                 $curBranch := b$
46:                 $B := B \setminus \{b\}$
47:             Recursively discard all branches in $B$
48:             Append $cmd$ to $trunk$
49:             $next := next + 1$
```

the ballot number, and the next agreement slot number extracted from the new branch identifier. This way, if the parent and the new configuration share the same proposer (which is a common case in practice), the command agreements can proceed to execute speculatively in the new branch under the same ballot, thus maintaining the same command throughput across the two configurations.

A speculative branch B is called *valid* if the reconfiguration command that spawned B is agreed in its parent branch. Conversely, B is *invalid* if the agreed command occupying B's parent branch slot at which B is rooted is either reconfiguration command for a different configuration, or a user command. (This way, each branch in $branches$ can be either one of the following three types: valid, invalid, or speculative.) The globally ordered command prefix is represented by the state variable called $trunk$ defined in line Algorithm 2.14. The valid branches can be merged into $trunk$ provided they form a continuous sequence rooted at a slot of $trunk$. This is accomplished by the background task shown in Algorithm 3, which traverses a series of valid branches (stored in the $curBranch$ variable) switching to the next branch whenever the reconfiguration command that spawned this branch becomes agreed and merged into the trunk (lines Algorithm 3.42–46). At this point, all other branches rooted at the same slot as well as the currently traversed parent branch are recursively discarded (lines Algorithm 3.44, 3.47).

The speculative branch structure is exposed to the clients, and the commands agreed in those branches with a special flag indicating that their final status is not yet known. In addition, the clients are notified whenever a new speculative branch is created. They can then act upon this notification, e.g., by creating the snapshot of the current state for the possible future rollback. The clients are delivered a *validate* notification for a command once it is merged into $trunk$, and an *invalidate* notification if the branch to which the command belongs has been discarded.

4.2 State Transfer

When the system transitions to a new configuration, the implementation must ensure that the latest system state is transferred to the members of the new configuration. Since the standard practice of interrupting the user command execution for the duration of the state transfer would cancel out the benefits of speculation, in FRAPPÉ, both state transfer and speculative command ordering proceed in parallel as explained below.

An instance of the state transfer protocol is activated whenever a new branch is created in response to a JOIN message received by a replica (line Algorithm 2.33). Since the instances of the state transfer protocol in all branches execute the same logic, in the following, we limit our attention to the steps taken by a single state transfer instance activated in response to changing the cohort membership from C to C'. Note that although the replicas in $C \setminus C'$ might eventually leave the cohort (i.e., become idle), they nevertheless remain available for serving state transfer requests as long as they are on line.

If both C and C' share the same leader, then it can continue ordering commands by executing only the second ("proposal") phase of the Paxos protocol using the same ballot number. Otherwise, the new leader must execute the first ("prepare") phase of Paxos to establish the new ballot number, and learn the latest global state prior to resuming the normal operation.

In addition, each newly joined process $p \in C' \setminus C$ (if any) determines which parts of $trunk$ of size n are not reflected in its local copy of the state, and initiates state transfer with the members of C to fill out the missing parts. If the replicas in C are unavailable or do not have up-to-date state (e.g., if state transfer in the C's branch is still in progress), p will request the missing information from the currently available members of the replication group. If p looses some of the ordering decisions reached in C' prior to p's join, it will request the missing commands directly from the C''s leader.

5 Conclusions

In this paper, we have introduced FRAPPÉ – a replication platform designed for supporting elastic services and applications hosted in clouds and large data centers. The key to the FRAPPÉ's efficiency is the new replicated state machine protocol, which is capable of avoiding the delays associated with the replication group reconfiguration by ordering commands speculatively in parallel to reaching agreement on the next stable configuration. We have discussed the FRAPPÉ's architecture, API, and core ideas underlying our speculative state machine implementation.

FRAPPÉ is being developed at IBM Research, and slated to be included into the future platform-as-a-service offerings as a foundational tool. The experimental performance study of the initial prototype are currently under way, and the results will be available in the full version of the paper.

References

[1] Amazon web services. [Online]. Available: http://aws.amazon.com.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.

[3] Y. Amir and C. Tutu. From total order to database replication. In *ICDCS*, pages 494–, 2002.

[4] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[5] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, pages 123–138, 1987.

[6] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[7] D. Dolev, I. Keidar, and E. Y. Lotem. Dynamic voting for consistent primary components. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 63–71, New York, NY, USA, 1997. ACM.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[10] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[11] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[12] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. Technical report, Microsoft Research, 2008.

[13] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, New York, NY, USA, 2009. ACM.

[14] 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 *In Proc. EuroSys06*, 2006.

[15] N. Lynch and A. A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *In DISC*, pages 173–190, 2002.

[16] R. D. Prisco, A. Fekete, N. A. Lynch, and A. A. Shvartsman. A dynamic primary configuration group communication service. In *DISC*, pages 64–78, 1999.

[17] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4:243–254, January 2011.

[18] A. Ricciardi and K. Birman. Process Membership in Asynchronous Environments. Technical Report TR93-1328, Cornell University, February 1993.

[19] J. Sussman and K. Marzullo. The bancomat problem: an example of resource allocation in a partitionable asynchronous system. *Theor. Comput. Sci.*, 291:103–131, January 2003.

[20] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.