

The SMART Way to Migrate Replicated Stateful Services

Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken,
John R. Douceur, and Jon Howell

Microsoft Research

{lorch, adya, bolosky, rchaiken, johndo, howell}@microsoft.com

Abstract

Many stateful services use the replicated state machine approach for high availability. In this approach, a service runs on multiple machines to survive machine failures. This paper describes SMART, a new technique for changing the set of machines where such a service runs, i.e., *migrating* the service. SMART improves upon existing techniques in three important ways. First, SMART allows migrations that replace non-failed machines. Thus, SMART enables load balancing and lets an automated system replace failed machines. Such autonomic migration is an important step toward full autonomic operation, in which administrators play a minor role and need not be available twenty-four hours a day, seven days a week. Second, SMART can pipeline concurrent requests, a useful performance optimization. Third, prior published migration techniques are described in insufficient detail to admit implementation, whereas our description of SMART is complete. In addition to describing SMART, we also demonstrate its practicality by implementing it, evaluating our implementation's performance, and using it to build a consistent, replicated, migratable file system. Our experiments demonstrate the performance advantage of pipelining concurrent requests, and show that migration has only a minor and temporary effect on performance.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems

General Terms: Algorithms, reliability

Keywords: Migration, replication, reconfiguration, Paxos, replicated state machine

1. INTRODUCTION

Increasingly, services are being designed to seamlessly tolerate machine failures so they can use inexpensive, unreliable hardware yet still be highly available. A common way to make a service tolerate machine failures is to repli-

cate it on several machines. However, replication can only mask a limited number of failures, and the longer the service runs the more likely the failure count will exceed this number. Therefore, a service must replace failed machines in a timely fashion, and this requires that the service be able to change its *configuration*, i.e., the set of machines replicating it. Changing the configuration, also called *migration*, has other purposes, e.g., moving replicas from highly loaded machines to lightly loaded ones, or changing the number of machines replicating the service. This paper presents the Service Migration And Replication Technique, a.k.a. SMART, our technique for migrating a replicated service.

It is easy to achieve consistency in a replicated service with no changing state, so this paper concerns only stateful services, such as file systems, databases, or market trading systems. A stateful service must replicate its state so that clients never see inconsistencies, even when failures occur.

The state of the art for building consistent, replicated services is the *replicated state machine* approach [13, 20, 22], so SMART targets services using this approach. In this approach, a *replica*, i.e., a copy of the service, runs on each machine; these replicas use the Paxos protocol to stay synchronized. System designers are increasingly using this approach for several reasons. First, it works on cheap and easily-administered hardware, such as PCs connected by Ethernet. It does not, e.g., require storage-area networks, RAID, or networks with partition-free or real-time guarantees. Second, its requirement of service determinism is becoming easier to satisfy with the advent of new techniques and tools [3, 21]. Third, the approach can be extended to deal with Byzantine server failures, i.e., failures that cause behavior other than stopping [3]. This is increasingly important as services move to less reliable infrastructures, such as on-demand computing systems and peer-to-peer overlays, and also as people increasingly exploit security weaknesses.

Existing migratable replicated state machine implementations have several restrictions that limit their adoption in a general setting [23]. First, they cannot perform migrations that remove or replace a non-failed machine. So, for instance, they cannot move a replica from a highly loaded machine to a less loaded machine, and they make it dangerous to rely on an imperfect failure detector, such as an autonomic system, to decide whether to replace a machine. Second, they cannot process requests in parallel, a useful performance optimization for services with concurrent requests. Third, important details of these systems have not been published, making it difficult for others to duplicate their designs. We are only aware of one publication de-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys '06, April 18–21, 2006, Leuven, Belgium.

Copyright 2006 ACM 1-59593-322-0/06/0004 ...\$5.00.

scribing a migratable replicated state machine, and it only describes its approach at a high level [16].

Consequently, we developed a new technique, SMART, for migrating replicated state machines. SMART can pipeline concurrent requests, and it allows arbitrary migrations. Most notably, it can safely perform a migration that replaces a machine, even if it is not certain that the replaced machine has failed. This ability enables autonomic service management.

A key feature of SMART is *configuration-specific replicas*. Each replica is associated with one and only one configuration, so a migration creates a new set of replicas, one for each machine in the new configuration. For example, when the service migrates from $\{A, B, C\}$ to $\{A, B, D\}$, SMART does not create a new replica on D and destroy the one on C, as current approaches do. Instead, it starts three new configuration-2 replicas on A, B, and D, and keeps the three configuration-1 replicas running on A, B, and C until the new configuration is established. This feature substantially simplifies the process of migration and thereby allows SMART to overcome the problems of existing approaches. However, it makes implementation inefficient, so we introduce the use of *shared execution modules* to eliminate this inefficiency.

We implemented SMART to demonstrate its practicality and to evaluate its performance. Using this implementation, called LibSMART, we built a file system that is fully consistent, replicated, and migratable. We also built a less complex service to enable microbenchmarks. We show that allowing pipelining reduces latency by 14% when there are multiple clients submitting requests concurrently. We also show that clients observe little additional latency due to migration. During migration, one or two client requests see about 25 ms of additional latency, and for a short while after migration, clients see about 0.4 ms of extra latency per request.

The contributions of this paper are as follows:

- We introduce the concept of configuration-specific replicas, which enables arbitrary migrations and pipelining of concurrent requests.
- We describe how to use shared execution modules to efficiently implement configuration-specific replicas.
- We describe SMART, our technique for migrating replicated state machines. We are the first to describe all details necessary to implement such migration.
- We demonstrate SMART's practicality by implementing it and evaluating that implementation.

For space reasons, we only sketch our arguments for SMART's correctness. Our technical report [12] contains a formal proof.

The paper is structured as follows. §2 provides background about Paxos and the replicated state machine approach. §3 describes current approaches to service migration and explains their limitations. §4 describes SMART, our technique for service migration. §5 sketches our arguments for SMART's correctness. §6 describes our implementation and our experimental evaluation of it. §7 describes related work, and §8 discusses avenues for future work. Finally, §9 concludes.

2. BACKGROUND: PAXOS

This section provides enough detail about Paxos and the replicated state machine approach to understand SMART. Other sources contain complete details [13, 14, 22].

2.1 Assumptions

First, we describe the assumptions that must hold for the replicated state machine approach to be applicable. Except where noted, we make the same assumptions for SMART.

We assume the service runs only on *fail-stop* machines, i.e., machines that fail only by stopping. In §8, we discuss how SMART could be extended to deal with Byzantine failures. Note that if a machine crashes and eventually recovers, this is temporary unavailability, not a failure.

Paxos assumes that fewer than half the machines will fail. In other words, there is always some future time when a *quorum* of replicas will be alive, where a quorum is a simple majority. SMART weakens this assumption in that once a new configuration is established, machines in older configurations can fail. After all, one purpose of service migration is to eliminate dependence on the current configuration when one believes it may stop operating properly. Consequently, SMART only assumes that fewer than half the machines in a configuration will fail before the next configuration is established.

The service must be *deterministic*, i.e., its state changes and outputs can depend only on its state, its input requests, and the order of those requests. So, for example, the service cannot use local random number generators or clocks, and must be single-threaded to avoid non-deterministic effects of thread scheduling. Many techniques and tools simplify building deterministic services [3, 21]. For instance, there are deterministic techniques to approximate the current time, schedule future events, and choose pseudorandom numbers. It may also be possible to use a virtual machine monitor to make arbitrary non-deterministic services run deterministically [11]. In our discussion of future work in §8, we sketch how SMART might be modified to allow non-deterministic services.

All we assume about network reliability is that if one alive process sends a message to another alive process an infinite number of times, it is eventually received.

2.2 Overview

In the replicated state machine approach [13, 20, 22], a replica, i.e., a copy of the service, runs on each of several machines. These replicas run the Paxos protocol to ensure they all execute the same client requests in the same order. Then, since the service is deterministic, the replicas change their states in the same way and produce identical outputs, e.g., replies to clients. The replicas thereby act like a single copy of the service that is more resilient to failure than any individual machine.

Paxos's goal is to ensure all replicas execute the same requests in the same order. In other words, Paxos must decide what request to execute first, what request to execute second, etc. In general, we say that Paxos *assigns* requests to *slots*, where the request it assigns to slot n is the one each replica will execute as its n th request.

Figure 1 illustrates how Paxos works. One of the replicas is the *leader*. To submit a request, a client sends it to this leader. The leader chooses an unused slot and sends each replica a *proposal*, which is a tentative suggestion that

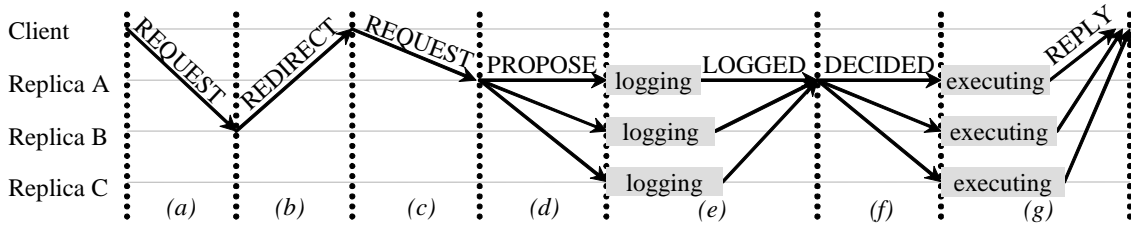


Figure 1: Example of Paxos operation. (a) The client sends a request to the replica it thinks is leader. (b) The client is wrong, so the recipient redirects the client to the correct leader. (Normally, the client is correct and thus skips steps a and b.) (c) The client sends its request to the correct leader. (d) The leader selects the next available slot and sends a message to each replica (including itself) proposing that the request fill the selected slot. (e) The replicas log the proposal, then notify the leader. (f) When the leader receives LOGGED messages from a quorum, it sends a message declaring the proposal decided. (g) Each replica receiving a DECIDED message executes the request and sends a reply to the client. The client ignores all but the first reply.

the given client request should occupy the given slot. Each replica receiving a proposal *logs* it, i.e., writes it to stable storage, then sends the leader a LOGGED message. Once the leader receives LOGGED messages from a quorum, it announces that the proposal has been *decided*, i.e., the request has been assigned to the slot.

One complication is that the leader may fail. So, the leader sends periodic HEARTBEAT messages to each replica, and a replica that does not hear one for a while elects a new leader by sending ELECT messages to each replica. Before a new leader does anything, it must learn enough about the actions of previous leaders to ensure it does not make conflicting assignments. So, it polls the replicas, asking what proposals they logged from previous leaders. Once it gets poll responses from a quorum of replicas, it can ensure it does not make conflicting assignments.

The Paxos leader change protocol includes other details that we have not described, but they are not relevant to understanding SMART. As we will see, SMART uses configuration-specific replicas, so the Paxos leader change algorithm runs only on static configurations and thus needs no modification.

A client may think a non-leader is leader, e.g., because of a recent leader change, but this gets corrected promptly. If a non-leader receives a request, it cannot propose the request so it replies with a REDIRECT message indicating the current leader. If the client does not receive a timely reply to a request, it broadcasts the request to all replicas.

Each replica executes requests in slot order. After executing the request in slot n , it waits to learn what request gets assigned to slot $n + 1$, then executes that request.

2.3 Practical implementation details

Checkpoints

If a replica loses its volatile state, e.g., due to a reboot, it may have to start over from the initial state and re-execute all requests. So, each replica periodically saves a *checkpoint*, i.e., a copy of its service state on stable storage. It restores its latest checkpoint after losing its volatile state.

We call the last slot executed before saving a checkpoint the *index* of that checkpoint. A replica discards a checkpoint only if it has a checkpoint with a higher index, so once a replica has a checkpoint with index n it will always have a

checkpoint with index at least n .

We say an index is *stable* once there is a quorum of replicas each having a checkpoint with that index or higher. If an index is stable, there will always eventually be an alive replica with a checkpoint with that index or higher. This is because there will always eventually be a quorum of alive replicas, and this quorum must overlap the quorum that made the index stable.

State transfer

Normally, a replica reaches the state following request n by executing requests 1 through n . However, it can sometimes accomplish this more expediently via *state transfer*, i.e., by receiving a copy of some other replica's checkpoint with index n . For instance, if a replica has only executed 100 requests, but receives a copy of a state checkpoint with index 200 from another replica, it can make this its current service state and thereby avoid executing requests 101 through 200.

Log truncation

Because stable storage is limited, we cannot force replicas to maintain every logged proposal forever. We thus require a way to let replicas discard old proposals from their logs, i.e., to perform *log truncation*.

Once n is a stable index, there will always eventually be an alive replica with a checkpoint with index n or more. This replica can transfer this checkpoint to any replica that has not yet executed request n , allowing the recipient to skip executing that request. This means no replica need ever know what request n is, so it is safe to discard logged proposals for slot n .

We can thus place a bound, `MaximumLogSize`, on the number of logged proposals a replica need ever hold in stable storage, by making the following rule. A leader never proposes a request to fill slot n until index $n - \text{MaximumLogSize}$ is stable. This way, a replica need not log a proposal for slot n until it can discard logged proposals for all but the preceding `MaximumLogSize` slots.

Limiting the log size can improve performance when replicas have NVRAM available. As long as the log fits in NVRAM, replicas can log proposals without expensive disk writes. This, in turn, speeds up the critical path of request handling.

3. MIGRATION: CURRENT METHODS

In this section, we discuss current approaches to migrating replicated state machines. §3.1 presents Lamport’s idea, which all current approaches use. §3.2 describes challenges in implementing this idea, and §3.3 shows how current implementations address them only by significantly restricting functionality.

3.1 Lamport’s idea

Lamport’s description of Paxos includes the following idea about how to perform migration [13]. The service state includes the configuration, and the service migrates when a request changes this configuration. The migration does not happen immediately; it takes effect α slots later, where α is some positive constant. In other words, if n is the slot of the request that changes the configuration, then this change takes effect starting with slot $n + \alpha$.

Using a small α can hurt performance, because the leader cannot make a proposal for slot $n + \alpha$ until it has executed slot n . Until it executes slot n , it cannot know whether the request in that slot excludes the leader’s machine from the configuration as of slot $n + \alpha$. For instance, if the leader has only executed slots 1–100 and α is 2, it can only propose slots 101–102 because as far as it knows it is excluded as of slot 103. The bigger α is, the less likely the leader must wait to propose a request until it has finished executing some other one, and thus the more undecided proposals the leader can have outstanding. We call the use of concurrent undecided proposals *pipelining*, reflecting how this overlaps the Paxos network delays for multiple requests.

3.2 Implementation challenges

Storing the configuration in the state is an elegant idea for migrating replicated state machines. However, it is not a complete solution to the problem. Implementing the idea involves addressing several challenges, including the following five.

Unaware-leader challenge: A new leader may not know the latest configuration.

A new leader must poll the replicas to ensure its assignments do not conflict with previous leaders’ assignments. However, when it is elected, it may have not yet executed a request that changed the configuration. It may then poll the wrong set of replicas, not learn about a recent assignment, and make a proposal that conflicts with that assignment.

For example, suppose that after the service migrates from $\{A, B, C\}$ to $\{A, B, D\}$, the leader, A, fails. Now, B initiates a leader change, but B has not yet executed the request that migrated the service. So, it only polls A, B, and C, and is content getting responses only from B and C, since they are a quorum of the configuration it knows. However, A and D together constitute a quorum of the new configuration, so they may have assigned some request to a slot. B does not learn of these assignments, and may propose requests that conflict with them.

Window-of-vulnerability challenge: Migrations that remove or replace a machine can create a period of reduced fault tolerance.

During migration, a service can have reduced fault tolerance. For example, a service replicated on three machines should be able to survive one machine failure. But, if it mi-

grates from configuration $\{A, B, C\}$ to $\{A, B, D\}$, a failure of A at the wrong time can halt the service forever. Suppose the leader, A, crashes while sending DECIDED messages for the request that changes the configuration. Only C receives a DECIDED message, and when it executes the request, it learns that it has been removed from the configuration and terminates. Now, some replica must become the new leader. B thinks the only replicas are A, B, and C, so it will never receive a quorum of poll responses; C has terminated; and D is not even aware yet that it is part of the configuration. So, no replica can become leader, and the service halts forever even though only one machine has failed.

It might seem we could avoid this problem by first migrating to $\{A, B, C, D\}$ and then migrating to $\{A, B, D\}$, but this also has a window of vulnerability. The leader, A, may crash while sending DECIDED messages for the request that removes C. Only C receives a DECIDED message, and it executes the request and terminates. Now, some replica must become the new leader. However, A has failed and C has self-terminated, so at most two replicas, B and D, can reply to polls. Since $\{B, D\}$ is not a quorum of $\{A, B, C, D\}$, no replica can receive a quorum of poll responses and become leader.

Extended-disconnection challenge: After a long disconnection, a client may be unable to find the service.

If a client reconnects after a long disconnection, its idea of the latest configuration may be out of date due to migrations during the disconnection. Furthermore, all machines in the configuration it knows of may have permanently failed. Thus, every machine it contacts will never respond, it will never learn of a working configuration, and it will not be able to submit requests.

Consecutive-migration challenge: If request n changes the configuration, requests $n + 1$ through $n + \alpha - 1$ cannot change the configuration.

Suppose request n removes machine C from the configuration as of slot $n + \alpha$, and request $n + 1$ restores C to the configuration as of slot $n + \alpha + 1$. Suppose also that C does not execute request n because it receives a state transfer of the checkpoint with index $n + 1$. In this case, it will never see the configuration that removes it from the configuration, and will incorrectly think it is responsible for slot $n + \alpha$.

Multiple-poll challenge: A new leader may have to poll several configurations.

It is possible for several undecided proposals to be outstanding at the time of a leader change. Thus, it is possible that some of these are managed by one configuration and others are managed by another. The leader change algorithm must be modified so it can poll multiple configurations.

3.3 Current approaches

Current approaches to migration address these challenges at the cost of restricted functionality. In this section, we discuss a typical example of these approaches, Petal’s global state manager (GSM) [16, 23]. Anecdotal evidence suggests that unpublished commercial systems use similar techniques.

To address the unaware-leader challenge, Petal’s GSM uses the following unpublished technique [23]. After a new

leader collects poll responses from a quorum of replicas, it fetches the configuration from the responder that executed the most requests. If this configuration is different than the leader’s, it updates its configuration and restarts the leader change process. Petal’s GSM also enforces two additional restrictions that, together with the preceding technique, are sufficient to address the unaware-leader challenge. First, requests are not pipelined. Second, only two types of migration are allowed: adding one machine, or removing one machine. This ensures quorums from consecutive configurations always overlap. Since SMART addresses the unaware-leader challenge in a different way, it can pipeline requests for performance and it does not require quorums from consecutive configurations to overlap. In fact, SMART does not require any overlap at all between consecutive configurations.

To sidestep the window-of-vulnerability challenge, Petal’s GSM removes a machine via migration only if a human knows it is failed and beyond repair. Thus, although such a migration reduces the number of tolerable permanent failures by one, this is acceptable because one machine is already known to have permanently failed. For instance, Petal’s GSM will only migrate from {A, B, C, D} to {A, B, D} when C is permanently failed. It is reasonable to not survive the failure of A during migration, since the service is not expected to survive two simultaneous machine failures. SMART addresses the window-of-vulnerability challenge differently, by eliminating the window of vulnerability. Thus, unlike Petal’s GSM, it can remove a non-failed machine via migration. In particular, it can migrate replicas from highly loaded machines to lightly loaded ones, and it can safely rely on an autonomic system with an imperfect failure detector to decide when to replace a machine.

Petal’s GSM addresses the extended-disconnection challenge similarly to how SMART does. We discuss how SMART addresses it in §4.5.

Petal’s GSM avoids the consecutive-migration and multiple-poll challenges by requiring $\alpha = 1$. This precludes pipelining concurrent requests. SMART uses a different approach, so it can pipeline concurrent requests.

4. SMART

This section describes SMART incrementally, adding details and optimizations as it proceeds. Figure 3, appearing later, will present an overview of the protocol this section describes.

4.1 Configuration-specific replicas

A key feature of SMART is *configuration-specific replicas*: each replica is associated with one and only one configuration. A migration request creates a new set of replicas, one for each machine in the new configuration. Figure 2 illustrates an example. When the service migrates from configuration {A, B, C} to {A, B, D}, we do not create a new replica on D and destroy the one on C, as current approaches do. Instead, we keep the three configuration-1 replicas running on A, B, and C, and start three new configuration-2 replicas on A, B, and D.

The old configuration’s replicas continue running even after creating the new configuration. They remain running long enough to ensure there is no period of reduced fault tolerance, i.e., to address the window-of-vulnerability chal-

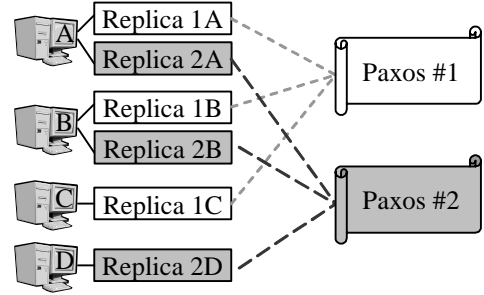


Figure 2: Example of configuration-specific replicas, where configuration 1 is {A, B, C} and configuration 2 is {A, B, D}.

lenge; we discuss this further in §4.4. While both configurations are running, if one machine is in both configurations, that machine will simultaneously run two replicas, one for each configuration.

Each configuration uses its own instance of Paxos. The replicas of that configuration are *cohorts* of each other, i.e., participants in the same Paxos instance. For example, each configuration’s Paxos has its own leader, one of the replicas in that configuration. Leaders of different configurations may or may not happen to be replicas on the same machine. Since each instance of Paxos has a static configuration, the leader change algorithm need not deal with a changing configuration or multiple simultaneous configurations. This straightforwardly addresses the unaware-leader and multiple-poll challenges.

We discuss how we address the extended-disconnection and consecutive-migration challenges in §4.5 and §4.9, respectively.

When a replica executes a request that creates a new configuration, it sends JOIN messages to each machine in this new set, telling them to join the configuration if they have not done so already. A machine joins a configuration by starting a replica associated with that configuration.

4.2 Avoiding inter-configuration conflict

Separate configurations use separate instances of Paxos, so we must ensure they do not assign different requests to the same slot. We achieve this by making each configuration responsible for a range of slots, *FirstSlot* through *LastSlot*, with no two ranges overlapping. If the request that creates a configuration is in slot n , then the new configuration’s *FirstSlot* is $n + \alpha$, and the old configuration’s *LastSlot* is $n + \alpha - 1$. Each replica only assigns and executes slots in its range because a leader only proposes slots in its range.

A leader can avoid proposing slots less than *FirstSlot* because each replica is created knowing its *FirstSlot*. If it is in configuration 1, it knows *FirstSlot* is 1. Otherwise, it was created due to a JOIN message, and the JOIN message specified this value.

It is trickier to avoid proposing slots after *LastSlot*, since often the leader will not know this value because it has not yet created a successor configuration. For this case, we use Lamport’s idea: the leader may not make a proposal for slot $n + \alpha$ until it has executed slot n . Thus, as usual, α controls the degree to which the leader’s proposals can get ahead of its execution.

For each server machine:

Whenever you learn that configuration n is defunct, and $n > d$ where d is the highest one you know is defunct,

Set $d := n$ and destroy all replicas in configurations $\leq n$.

Upon initial startup, if you are in the initial service configuration, Start a replica in configuration 1.

Upon receipt of a JOIN message for a configuration $n > d$,

Start a replica in configuration n if one isn't running already.

Upon receipt of a FINISHED, HEARTBEAT, or ELECT message addressed to a replica in configuration $n > d$,

If there is no local running replica in that configuration,

Reply with a JOIN-REQUEST message for configuration n .

Upon receipt of a client request to a replica in configuration $n \leq d$,

Reply with a NEW-CONFIGURATION message.

Upon receipt of any other message to a replica in configuration $n \leq d$,

Reply with a DEFUNCT message.

Upon receipt of a DEFUNCT message regarding configuration n ,

Note that configuration n is defunct.

For each replica on each server machine:

After executing a request that creates a new configuration, Send a JOIN message to each machine in that configuration.

After executing the last slot of your configuration,

From now until you are eventually destroyed, periodically send a FINISHED message to each machine in the next configuration.

Upon receipt of a FINISHED message,

If you have a checkpoint at or after your starting state, Reply with a READY message;

Otherwise, if you don't have your starting state,

Request your starting state from the sender.

Upon receipt of READY messages from a quorum of your successor configuration,

Note that your configuration is defunct. (This will cause you to be destroyed.)

Upon receipt of a JOIN-REQUEST message,

Reply with a JOIN message for the requested configuration.

Figure 3: Overview of the SMART protocol

4.3 Transfer of responsibility

We now discuss how a new configuration begins receiving and executing client requests.

When a leader has filled up all slots through `LastSlot`, it cannot make any more proposals. So, it responds to client requests with a `NEW-CONFIGURATION` message, telling the client to start using the new configuration. The client ignores the message if it already knows about a later configuration.

Before a new replica executes any requests, it must initialize its service state to an appropriate *starting state*. Since the first request this replica can execute is the one in slot `FirstSlot`, a proper starting state is a state reflecting the execution of at least slots 1 through `FirstSlot - 1`. Thus, it can acquire its starting state from a replica from the previous configuration that has executed its `LastSlot`, i.e., a *finished* replica from the previous configuration. In some cases, it may acquire its starting state from a cohort instead.

4.4 Destroying defunct replicas

In §3.2, we showed how a window of vulnerability arises when a replica terminates before the next configuration is able to make independent forward progress. In SMART, we eliminate this window of vulnerability by not destroying replicas of an old configuration immediately. In this subsection, we discuss when SMART can destroy a replica of an old configuration.

We say a configuration is *established* once `FirstSlot - 1` is a stable index of that configuration. After this, it will always eventually have an alive replica with a copy of the state reflecting slots 1 through `FirstSlot - 1`, obviating the need for any information about those slots. Since these slots are exactly those that previous configurations are responsible for, those configurations and their replicas are *defunct*, i.e., safe to destroy.

The replicas of a configuration must continue operating until a successor configuration is established. Recall from §2.1 that we assume a quorum of replicas in a configuration will continue operating at least until a successor is established. The liveness of a quorum enables that configuration

to complete its work, namely to eventually assign a request to `LastSlot`, execute that request, then propagate the resulting state to a successor configuration so that configuration can become established.

We use the following two-part protocol to ensure that a defunct, alive replica eventually destroys itself. First, when a machine receives a message addressed to a replica that has destroyed itself, it responds with a `DEFUNCT` message. A replica destroys itself if it hears a `DEFUNCT` message indicating a cohort or a member of a successor configuration has destroyed itself. Second, when a replica is finished, it creates a thread that periodically sends a `FINISHED` message to each replica in its successor configuration. This message asks the recipient to reply with a `READY` message if it has saved a checkpoint at or after its starting state. Once the finished replica has received a `READY` reply from a quorum of successor replicas, or one `DEFUNCT` reply, it knows some successor configuration is established and destroys itself.

When a replica destroys itself, its machine must remember its configuration number and successor configuration so it can send `DEFUNCT` and `NEW-CONFIGURATION` messages as appropriate. As an optimization, a machine only remembers this information for the highest-numbered configuration. It can then send `DEFUNCT` or `NEW-CONFIGURATION` messages on behalf of any lower-numbered configurations, which are necessarily also defunct. The latter message may inform a client of a successor configuration that is not the immediate successor of the one the client was addressing, but this poses no problem.

4.5 Configuration repository

Next, we discuss how we address the extended-disconnection challenge. If a machine reconnects after a long disconnection, it may not know the latest configuration due to migrations during the disconnection. Furthermore, due to the weak reliability assumption in §2.1, all servers it knows of may have permanently failed, so it may never discover a working configuration. If it is a client, its requests may never get executed. If it has a defunct replica, that replica may never learn it is defunct. This is a fundamental problem for any self-migrating service, and necessitates a *configura-*

tion repository where the service can store its most recent configuration information.

We use the repository as follows. Periodically, e.g., every five minutes, a leader of an established configuration writes its configuration information to the repository. A process suspecting a configuration is defunct reads the repository to try to learn a newer configuration. Also, a leader reads the repository before writing its own information; if it finds an earlier configuration, it sends DEFUNCT messages to the replicas in that configuration.

This protocol ensures correct operation even if the repository is not consistent or durable. Furthermore, the service can be highly available even if the repository is less available, because the repository is only necessary on those rare occasions when a machine reconnects after an extremely long disconnection. It is important that the repository need not be consistent, durable, and highly available, since otherwise we could not build a service with those properties unless we already had a repository service with those properties.

Because there are such weak requirements for the repository, there are several simple ways to build it. For instance, one can store the configuration information in a DNS entry.

4.6 Ensuring new replicas are created

Next, we discuss how we ensure that every alive machine in a non-defunct configuration eventually joins that configuration.

We cannot rely on the JOIN messages replicas send when they create a new configuration. There are a finite number of these messages, so they may all be lost. We can use them as an optimization to start the replicas quickly, but not as a guarantee that every machine that should join the configuration does so.

If a machine receives a FINISHED message addressed to a not-yet-created replica on that machine, it creates the addressed replica. Since some replica will send FINISHED messages repeatedly until it receives a READY message from a quorum, a quorum of the new configuration will eventually join. This does not completely solve our problem, since we want *every* alive machine to eventually join. So, in addition, if a machine receives a HEARTBEAT or ELECT message addressed to a not-yet-created replica, it creates the addressed replica.

A small problem with this is that a machine must join upon receiving a FINISHED, HEARTBEAT, or ELECT message, but such a message does not specify the configuration's set of machines or FirstSlot. Thus, a machine does not join immediately upon receiving such a message. It sends back a JOIN-REQUEST message, and the recipient replies with a JOIN message.

4.7 Acquiring starting state

Recall that a new replica generally obtains its starting state from a finished replica in its predecessor configuration. Instead of repeatedly polling those replicas, waiting for one to be finished, a new replica just waits to receive a FINISHED message, then asks the sender for a state copy.

However, there is a wrinkle. Once the predecessor configuration becomes defunct, its replicas may destroy themselves and become unable to provide a state copy. So, sometimes a new replica must acquire its starting state from one of its cohorts. A replica can always eventually do this, because if its predecessor configuration is defunct then its configuration

is established, and eventually an alive cohort can provide a state copy with index at least FirstSlot - 1.

4.8 Null requests

In Paxos [13], sometimes upon election a new leader must propose a *null request* to fill a slot. A null request is simply an extra request whose execution does nothing, so it is always safe to propose a null request. SMART has the leader submit null requests in an additional scenario.

When a leader knows its LastSlot, it proposes null requests for all remaining unproposed slots. This hastens establishment of the successor configuration, reducing the time that correctness relies on the old configuration's machines and letting replicas of the old configuration destroy themselves sooner. It also serves to prevent deadlock, as follows.

Suppose a leader with LastSlot of 100 has made proposals for slots 1-100. Having filled up all its slots, it starts redirecting clients to the next configuration as discussed in §4.3. Now, the leader crashes, and the network loses all its proposal messages for slots 98-100. A new leader is elected, learns about slots 1-97 in its poll, and waits for a client request to propose for slot 98. However, it waits in vain: all clients were redirected to the next configuration, so they will not submit a request to that leader. This next configuration may assign requests to slots 101 and beyond, but not slots 98-100. No request ever gets assigned to slot 98, so the service stops forever. Our technique avoids this scenario since the newly elected leader will propose null requests for slots 98-100, allowing the service to make progress.

4.9 Configuration information in state

Often, a replica needs to obtain from its service state information about its successor configuration: whether it has been created and, if so, its FirstSlot and its set of machines. A leader needs this to determine for what slots it may propose. A replica needs this to determine whether it is finished and, if so, where to send FINISHED messages.

The consecutive-migration challenge arises because sometimes this information may not be present, having been overwritten by a later configuration. However, in SMART, the service state includes information not just about the latest configuration, as Lamport recommended, but also about older, non-defunct configurations. Consequently, the consecutive-migration challenge does not arise, and SMART can let any request change the configuration.

The service state may also include information about defunct configurations. However, the service should eventually discard such information to prevent its state from growing indefinitely. Since it is deterministic, it can only change its state while executing requests. So, at the beginning of executing request n , it checks whether $n - \text{MaximumLogSize}$ is the LastSlot of some configuration C . If so, it knows that C is defunct, and discards information about it. It knows C is defunct because, as we discussed in §2.3, no leader can propose request n unless $n - \text{MaximumLogSize}$ is a stable index of its configuration. This means that LastSlot of C is a stable index of some successor configuration, so C is defunct.

4.10 Shared execution modules

The scheme described thus far is inefficient in that a new replica must always copy its starting state from another replica. For many services, this state can be quite large, so copying it from one machine to another, or even just copy-

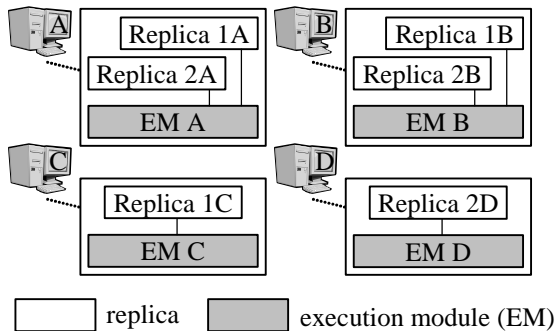


Figure 4: This figure illustrates shared execution modules, i.e., that all replicas on a machine share a single EM. Here, configuration 1 is $\{A, B, C\}$ and configuration 2 is $\{A, B, D\}$.

ing it between processes on the same machine, can be time-consuming. In this subsection, we describe an optimization we call *shared execution modules* that obviates most copying and saves space.

The optimization factors some of the functionality out of each replica into a separate module called an *execution module* (EM). This functionality includes storing service state, modifying state by executing requests, and saving and transferring state checkpoints. The remaining functionality remains in the replica, including acting as leader, logging proposals, and electing a new leader. This way, we can share a single EM among all replicas on the same machine, as illustrated in Figure 4.

Shared EMs reduce state copying as follows. Frequently, successive configurations will overlap, so several new replicas will be colocated with replicas of the old configuration. For instance, an autonomic system might replace a failed machine C in configuration $\{A, B, C\}$ by reconfiguring to $\{A, B, D\}$, so the new replicas on A and B find themselves with colocated replicas. Each new replica that is colocated with a replica from its predecessor configuration defers copying its starting state. It hopes that the colocated replica will soon finish executing its final slot, thereby putting the EM in exactly the state the new replica needs and obviating a copy.

However, its hopes may be dashed, and the colocated replica may not reach its final state before destroying itself. Fortunately, even in this case there is opportunity to save state-copying costs. Although the EM did not execute all requests from the previous configuration, it likely executed most of them, so its state is close to the required starting state. Thus, the replica can substantially reduce copying time with an incremental state transfer [3].

Some things a replica used to do internally now involve communication with the EM. For instance, it no longer executes requests. Instead, it tells the EM when requests are assigned to slots, and the EM executes those requests when it is ready.

A seemingly problematic aspect of EMs is that a replica may find its EM state progressing beyond `LastSlot`. For instance, if replicas in configurations 3 and 4 coexist on a machine, then the replica in configuration 4 may cause the EM to execute a slot that configuration 4 is responsible for. Or, the replica in configuration 4 may overwrite the EM state

with a checkpoint copy reflecting slots that configuration 4 is responsible for. However, this does not pose a problem, since the only reason a replica ever inspects its EM’s state is to obtain information about its successor configuration. As we discussed in §4.9, executing further requests cannot change this information due to the way we manage configuration information in the state.

5. CORRECTNESS

In this section, we sketch our arguments about SMART’s correctness. For the full proof, see our technical report [12], which specifies our system and proves its safety property in the formal systems-specification language TLA+ [15].

SMART’s safety property is that no two replicas ever assign different requests to the same slot.

Basic Paxos uses the following proof. Suppose two leaders assign requests to the same slot. The earlier leader must have gotten a quorum of replicas to log its corresponding proposal, and the second leader must have gotten a quorum of replicas to reply to its poll so it could make any proposal at all. Quorums overlap, so the second leader must have heard about the first leader’s logged proposal and thus proposed the same request for the slot. So, both leaders assign the same request to the slot.

However, this proof does not work for SMART, since not all quorum pairs overlap. For instance, one quorum might be $\{A, C\}$ from configuration $\{A, B, C\}$ and another might be $\{B, D\}$ from configuration $\{A, B, D\}$.

We start by proving the following lemma: if only one configuration is responsible for a slot, only one request is assigned to that slot. Configuration-specific replicas make this proof straightforward. A replica only sends Paxos messages to replicas of the same configuration. Therefore, if only one configuration is responsible for a slot, the quorum of replicas that logs a proposal for that slot must come from the same configuration as any quorum that replies to a poll that enables a future leader to create a new proposal for that slot. Reasoning as in the basic Paxos proof, the lemma follows.

We now prove that only one request is assigned to each slot by induction on slot. Only the initial configuration is responsible for slot 1, so by the lemma, the induction condition holds for slot 1. Now, suppose it holds for slots 1 through n . This implies only one request is assigned to each of those slots. The state machine is deterministic, so the outcome of executing requests 1 through n will be the same everywhere such an outcome is observed. This outcome specifies the configuration responsible for slot $n + 1$, and the uniqueness of that configuration means only one request will be assigned to that slot according to the lemma. This completes the inductive step and thus the proof.

The proof in the companion technical report [12] is formal and far more detailed. One illustration of this is that it showed us a bug in both the implementation and the protocol specification. Our specification considers volatile state by admitting a “crash” action that erases certain variables. We discovered that after a crash, a leader forgot which slots it had proposed for. If it recovered quickly enough that none of its cohorts noticed its failure, it could remain leader throughout its crash and recovery. Then, it could propose new requests in slots it had already used, possibly leading to conflicting requests for the same slot. Once we discovered this bug, it was easy to fix.

6. EXPERIMENTAL RESULTS

To evaluate SMART, we built a prototype implementation of it and performed experiments to evaluate its performance. This section describes that implementation, the methodology of our experiments, and our experimental results.

6.1 Implementation

We implemented SMART as a library called LibSMART. Service implementations can use this library to obtain replication and migratability. LibSMART presents a similar interface to BFT [3], enabling us to port existing services written for that interface. LibSMART’s implementation of SMART is complete except for the configuration repository.

To demonstrate SMART’s practicality, we used LibSMART to build a real service: a consistent, replicated, migratable file system. We ran thousands of hours of file system traces on this new file system, and thousands of runs that verified the correctness of our implementation during disconnections, reboots, and migrations. Furthermore, building this service was fairly straightforward. We already had a file system that used BFT for consistency and replication [1], so by porting it to LibSMART we achieved migratability. Note that by doing so we replaced its Byzantine fault tolerance with mere fail-stop fault tolerance; it is future work to enable Byzantine fault tolerance in SMART.

The file system uses highly effective techniques to hide the latency of server operations, so evaluating its performance sheds little light on the performance of LibSMART. Therefore, in this section, we benchmark LibSMART with a simple key/value service. This service permits client requests that read, write, and delete string values associated with integer keys. Each replica of the service caches information in memory for the most recently accessed 1,000 keys. Thus, it must access the disk when processing a request for a non-recently accessed key.

6.2 Methodology

Our experimental test bed uses seven identical HP D530 convertible mini-towers, each with a 3.2 GHz Intel Pentium 4 processor with 800 MHz Front Side Bus, 512 KB L2 cache with HyperThreading enabled, 1 GB 400 MHz DDR Dual Channel RAM, and 80 GB 7200 RPM PATA hard drive. Each runs Windows XP Professional with Service Pack 2. Each is connected to the same LAN subnet via built-in 100 Mb/s Ethernet. This subnet provides sub-millisecond ping times.

The first machine runs service clients. The next three, which we call A, B, and C, act as servers in the service’s initial configuration, with A as the initial Paxos leader. Two other machines, which we call D and E, act as servers in one experiment that requires five machines instead of three in the initial configuration. The last machine acts as a replacement during migration; accordingly, we call it R. We coordinate experiments using a host machine on the same subnet.

For most experiments, the only client requests we measure are ones that read a key expected to be in each server’s cache. We chose these requests since servers can execute them quickly. This highlights the more interesting sources of latency, e.g., those due to replication and migration.

Our test bed uses Seagate Barracuda ST380011A hard drives. Since these drives’ internal write caches are volatile, write caching could prevent SMART’s checkpoints and

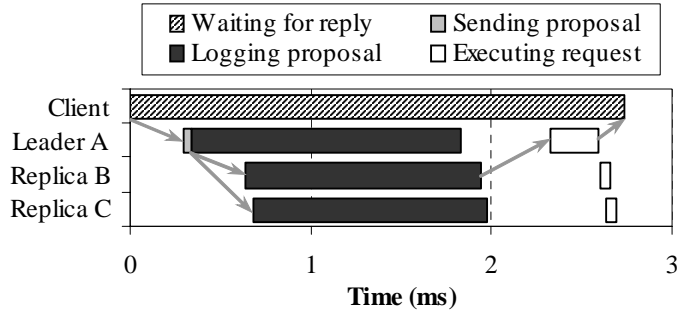


Figure 5: Timeline of one request selected for illustration

logged proposals from surviving a machine crash. Therefore, in a real deployment we would have to disable these disks’ write caches. However, since real servers’ hard drives generally use NVRAM write caches, and this will be increasingly true in the future, for most of our experiments we simulate the presence of such NVRAM by enabling write caching on the disks.

6.3 Request timeline

The goal of our first experiment is to present a timeline of the interesting events contributing to the latency of a single client request. This timeline will help frame further experimental results. To provide correspondence between times measured on different machines, we synchronize the machines’ clocks by broadcasting reference Ethernet packets [7].

Figure 5 shows the timeline that results from a single request we selected for illustration. The client sends a request, and after some communication delay the leader receives it and sends a proposal. The leader logs this proposal immediately, while the other replicas must wait until they receive it. Once the leader receives LOGGED messages from a quorum, it begins executing the request. In this case, the quorum consists of the leader and machine B, since machine C takes longer to receive and log the proposal. The leader sends a reply to the client, which considers the request complete. Machines B and C eventually hear the leader’s decision and execute the request, but this is not on the critical path seen by the client. Note that the leader takes longer to execute than the other replicas, because as an optimization in our implementation, only the leader sends a reply.

6.4 Effect of NVRAM

In the next experiment, we measure the effect of using NVRAM for disk write caching. A client submits 20,000 consecutive requests and we measure the average latency. We do this once with disk write caching on, simulating the presence of NVRAM, and once with disk write caching off.

We find that the 95% confidence intervals for average latency are $2.98 \text{ ms} \pm 0.04 \text{ ms}$ with NVRAM and $8.67 \text{ ms} \pm 0.06 \text{ ms}$ without it. We conclude that NVRAM reduces latency by approximately 5.7 ms. Figure 6 breaks down the latency into major components, showing, as expected, that the main reason for this difference is that logging takes longer without NVRAM.

Most of the remaining time is spent in communication, which includes waiting for network transmission, waiting for

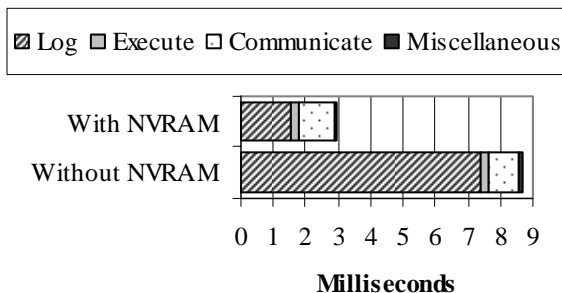


Figure 6: Breakdown of average request latency seen by client. “Miscellaneous” includes time spent proposing requests and time spent waiting for checkpoints.

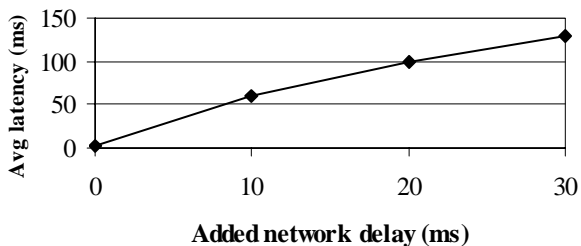


Figure 7: Effect of additional network delay on average request latency seen by client

message handlers to be scheduled, and encrypting and decrypting messages. Executing requests is minor, because we have chosen requests that are quickly executed. Incidentally, our implementation delays execution of a request while saving a checkpoint, which happens after every 50th request; repairing this is future work. However, in this example, the state changes are minor, so requests spend almost no time waiting for checkpoints.

6.5 Effect of network latency

Our next experiment measures the effect of network delay on client latency. A client submits 20,000 consecutive requests and we measure the average latency. We do this four times, each time simulating a different additional network delay. The added delays range from 0–30 ms.

Figure 7 shows the results. We see that for every 1 ms added to network latency, request latency goes up by approximately 4 ms. This is as expected since, as illustrated in Figure 5, there are four network hops on the critical path of handling a client request: the client sends the request to the leader, the leader sends a proposal to the other replicas, the replicas send the leader a LOGGED message, and the leader sends the client a reply.

6.6 Concurrent requests

Our next experiment measures the effect of pipelining on the latency of concurrent requests. Two clients run simultaneously, each submitting a stream of consecutive requests. We measure the average latency of 20,000 consecutive requests on one of the two clients. We run this experiment twice, once with pipelining enabled and once with it disabled. We disable pipelining by setting α to 1 instead of its default 10.

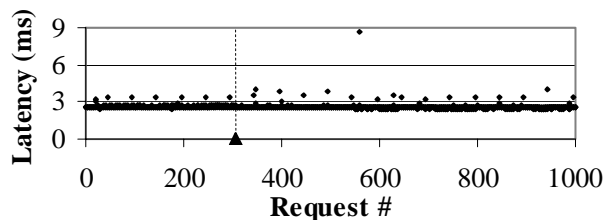


Figure 8: Request latencies seen by client before and after disconnection of machine C, shown as a vertical dashed line

With pipelining enabled, average latency is 4.3 ms; without pipelining, it is 4.9 ms. We see that in both cases the submission of requests by another client reduces performance. However, this effect is substantially reduced by pipelining. In all, pipelining reduces latency by 14%. We conclude that SMART’s ability to pipeline is useful in reducing request latency when there are concurrent requests.

Note that our implementation does not batch multiple requests, which would normally mitigate the effect of lack of pipelining. However, this is irrelevant for this experiment, since with two clients at most one request is waiting for proposal at a time and thus there is no opportunity for batching. Batching would reduce further performance degradation from a third client, a fourth client, etc., but it cannot help with the performance degradation from the first additional client.

6.7 Disconnection

We next perform a simple experiment to demonstrate resilience to a single failure. A client submits 1,000 requests; partway through, we disconnect machine C. Figure 8 shows the results, with the following two things most notable. First, there is no pause in service operation at the time of disconnection, since Paxos performs no special processing to exclude a disconnected non-leader machine. Second, average latency is the same before and after disconnection: 2.6 ms in both cases. The critical path of request handling involves the time it takes the fastest quorum of replicas to log the proposal and respond to the leader, so it goes essentially just as quickly with machine C disconnected as with all machines alive.

In another experiment, we disconnect the leader instead. The service continues operating correctly, but in this case the performance effect is far higher: after the disconnection, the next client request has a latency of 3.1 sec. This large delay is mostly due to the tunable 3-second timeout before another replica concludes the leader has failed and initiates a leader change.

6.8 Migration

In this section, we discuss experiments demonstrating the performance impact of service migration in our implementation. Specifically, we show the extra request latency that clients see during a service migration, and examine the sources of that additional latency.

To set up the first experiment, our client writes various 100-byte values to 10,000 keys. Then, the client submits a stream of read requests. Partway through this stream, another client running on the host machine issues an administrative request to migrate the service to configuration

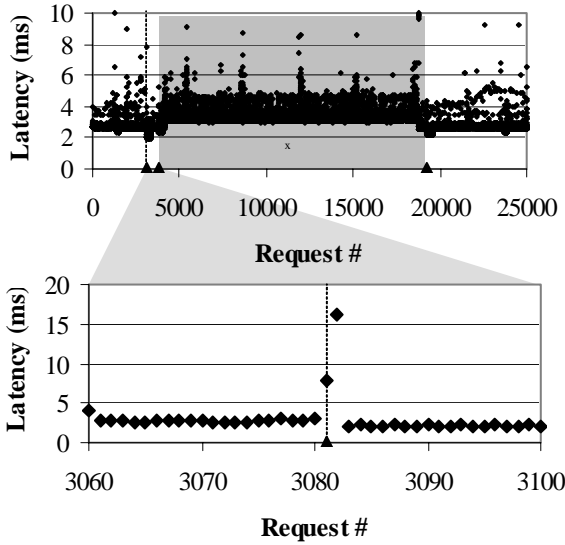


Figure 9: The top graph shows request latencies before and after migration of the service from $\{A, B, C\}$ to $\{A, B, R\}$. The migration is shown as a vertical dashed line, and the period during which machine B was copying state to machine R is shown with a gray background. For scale, several uninteresting outliers above 10 ms are not shown. The bottom graph shows the period surrounding the migration.

$\{A, B, R\}$. Figure 9 shows the request latencies observed by the reading client throughout this process.

The first two requests following migration incur a higher latency than typical, 8 ms and 16 ms. We explore the cause of this in §6.9. After those, request latencies return to the typical amounts seen before migration. Then, a short while later, there is a period of increased average latency, up to about 3.4 ms from the typical 3.0 ms. During this time, machine B takes slightly longer to process proposals as in the background it is sending a checkpoint to machine R. This takes about 53 seconds, mostly because our state transfer code has not been tuned for performance. After this transfer, latency returns to normal for the rest of the run. Overall, we conclude that migration has only a minor and temporary effect on observed latency, demonstrating the efficiency of our migration technique and implementation.

In our next experiment, we show that the delay caused by the checkpoint transfer can be eliminated by replicating the service on five machines instead of three. This experiment is just like the previous one except that the initial configuration is $\{A, B, C, D, E\}$ and the subsequent configuration is $\{A, B, R, D, E\}$. Figure 10 shows the results of this experiment. We see that, indeed, using a five-machine configuration eliminates the period of slightly elevated latencies during the gray region representing when the new machine is copying a checkpoint. The reason for this is as follows. When we replace one of five machines, four machines from the original configuration are also in the subsequent configuration. While one of these four is transferring state to the new machine, three machines are unaffected. These three machines by themselves constitute a quorum of the configuration, so it is their speed that determines the latency seen by clients.

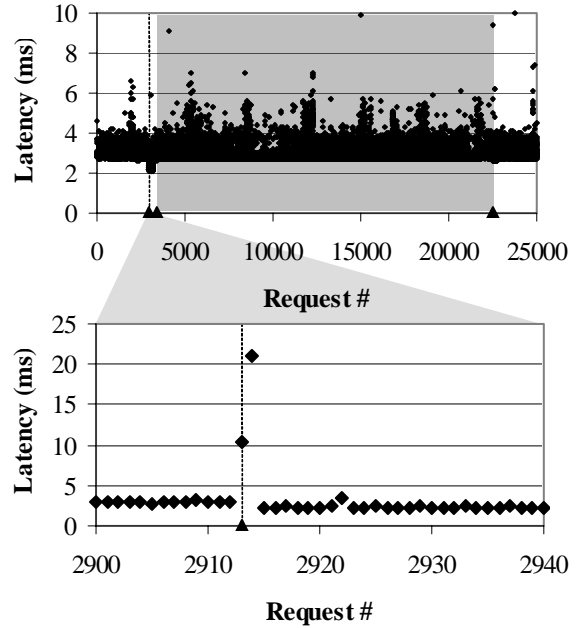


Figure 10: The top graph shows request latencies before and after migration of the service from $\{A, B, C, D, E\}$ to $\{A, B, R, D, E\}$. The migration is shown as a vertical dashed line, and the period during which machine D was copying state to machine R is shown with a gray background. For scale, uninteresting outliers above 10 ms are not shown. The bottom graph shows the period surrounding the migration.

Incidentally, the two client requests immediately following migration have elevated latencies of 10 ms and 21 ms, similar to what we observed in the previous experiment.

6.9 Sources of delay following migration

To understand the source of the latency increases immediately following migration, we conduct an additional experiment. In this experiment, we perform 200 consecutive migrations, alternating between configurations $\{A, B, C\}$ and $\{A, B, R\}$. We find that the delays observed by the client correspond to certain phases leader machine A goes through during migration, shown in Figure 11. Phase 1, taking 3 ms on average, is waiting for the migration request to be logged. Phase 2, taking 4 ms on average, is executing that request, including sending JOIN messages and proposing null requests. Phase 3, taking 12 ms on average, is waiting for null requests to be logged and then executing them. Phase 4, taking 8 ms on average, is waiting for a checkpoint save; our implementation always saves a checkpoint upon reaching a configuration’s final state, to hasten the establishment of the next configuration. Depending on whether any client request arrives during phase 1, either one request observes the latency of phase 2 and another observes the latency of phase 4, or a single request observes the latency of most of phases 2–4. The former case happened in our earlier migration experiments, explaining the two slightly elevated latencies immediately following migration. In our 200 migrations, the former case happens 83% of the time and when it does, the two delayed requests have average latency of 11 ms and

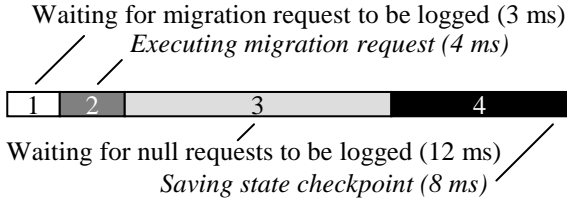


Figure 11: This rough timeline shows the phases leader machine A goes through during a migration. The two main sources of latency observed by clients are described in italics.

17 ms. The latter case happens 17% of the time during our 200 migrations, and then the average latency of the one delayed request is 28 ms.

Incidentally, if our implementation did not force requests to wait to execute during checkpoint saves, we would not see the latency of phase 4. Instead, we would see additional latency due to client redirection. Recall that a client request submitted after migration must incur two extra network transmission delays: one for the old leader to send a REDIRECT message to the client and one for the client to resubmit its request to the new leader. In our experiments, since requests were delayed until the end of phase 4, this redirection latency was not on the critical path, and so contributed nothing to observed latency.

7. RELATED WORK

Yin et al. [25] argue for the separation of agreement from execution in Byzantine fault-tolerant state machine replication, to reduce the number of execution modules in a static configuration and to enable the use of a privacy firewall. SMART also separates agreement from execution, but for a different purpose, namely to share one execution module among replicas of different configurations on the same machine.

Our use of configuration-specific service replicas is similar to the use of configuration-specific object replicas in systems such as RAMBO [17]. We are the first to use configuration-specific replicas to migrate replicated state machines, and to use shared execution modules to make this approach efficient.

Researchers have developed several methods for migrating replicated services that do not involve migrating replicated state machines. In the remainder of this section, we discuss these methods and compare them to SMART.

One method is to use a view-oriented group communication system (GCS) such as ISIS [2], Transis [6], or Horus [8]. Such a system allows a process to send a message to a group of processes, and allows the set of processes in this group, called the view, to change. This is a useful building block for replicated state machines, since it allows client processes to send requests to a changing group of servers. Most GCSes also provide *virtual synchrony*, meaning that processes in consecutive views see the same set of messages in the earlier view [4]. This simplifies coordination of replicas when views change. The main advantage SMART has over GCSes for replicating services is that Paxos deals more efficiently with temporarily unavailable machines. A GCS must incur the overhead of view change whenever any machine becomes temporarily unavailable [10, 17], while Paxos incurs compa-

rable overhead only when the leader becomes temporarily unavailable or the configuration changes.

Another method for building replicated services is to build them out of replicated objects, as in RAMBO [17] and Martin et al.’s reconfigurable Byzantine quorum system [19]. In such a system, an object is replicated among several servers, and the set of servers can change. A client request can either read or write an object. The main disadvantage of this approach is that some services cannot be composed out of read/write objects. For instance, some services need to atomically read and conditionally modify an object, or to atomically modify two objects. This is why a file service built on a replicated object system, such as Om [26], cannot provide namespace operations such as atomically moving a file out of one directory and into another.

Finally, a service can obtain migratability by using a separate configuration service, as done by Boxwood [18], GFS [9], and chain replication [24]. In this method, a separate service determines the current configuration of the main service. This approach is reasonable, but still requires a mechanism like SMART to allow the configuration service itself to migrate. For instance, most components in the Boxwood system are migratable, with the notable exception of the master configuration service, a limitation acknowledged by its authors [18]. SMART would be an excellent tool for building a migratable master configuration service for Boxwood.

8. FUTURE WORK

In this section, we discuss two promising avenues of future work. One is modifying SMART to allow non-deterministic services. The other is modifying it to survive a limited number of Byzantine server failures, i.e., failures that cause behavior other than merely stopping.

For some services, determinism is impractical, such as a multi-threaded service whose behavior is affected by thread scheduling. A standard way to deal with non-determinism in Paxos is *semi-passive replication* [5]. In this method, a leader does not propose requests. Instead, it tentatively executes requests, recording state changes and buffering outputs. What it proposes are those state changes and outputs. When another replica learns a proposal is decided, it merely applies the state changes. Thus, only one replica executes requests, as is appropriate considering the service is non-deterministic and thus may execute differently on different replicas. We believe SMART can be used with semi-passive replication, thereby enabling non-deterministic services.

There are well-known approaches, such as BFT [3], that let replicated state machines operate correctly despite limited Byzantine behavior. In principle, SMART could use BFT instead of basic Paxos, and thereby also tolerate such failures. However, many changes to SMART would be necessary. For instance, we would require a method like Martin et al.’s forgetting protocol [19] to ensure that old configurations cannot mislead clients if they become faulty.

9. CONCLUSIONS

In this paper, we presented SMART, our technique for migrating replicated stateful services. Unlike similar approaches, SMART can overlap processing of multiple requests, allows migrations with no overlap between consecutive configurations, and allows migrations that remove or replace a non-failed machine. Thus, it can migrate services

to balance load, and it can safely rely on autonomic systems to decide when to migrate. Also, we are the first to publish full details of how our migration technique works.

A key element of SMART is its use of configuration-specific replicas. We create a new configuration by starting a new replica of the service on each machine in the new configuration. These replicas run concurrently with the replicas from the old configuration until the new configuration is established. The replicas of each configuration run a per-configuration instance of Paxos; this simplifies the protocol since Paxos never has to run across multiple configurations simultaneously.

When multiple replicas run on the same machine, there is unnecessary duplication of service state, which can lead to expensive copying of service state across machines and/or processes. Thus, we use shared execution modules to let replicas on the same machine share service state.

We evaluated the performance of our technique by running experiments on our implementation of it. We found that SMART's ability to overlap processing of multiple requests reduces latency of requests when there are concurrent requests. We also found that migration has only a small and temporary effect on performance.

SMART allows a service to migrate and still preserve absolute consistency. Any deterministic algorithm may guide this migration, permitting any desired mechanism for balancing load and for recovering from lost fault tolerance due to failures. Even if the algorithm sometimes incorrectly decides that a machine has failed and needs replacement, it does not risk halting the service forever. Such autonomic migration is an important step toward full autonomic operation, in which administrators play a minor role and need not be constantly available.

10. ACKNOWLEDGMENTS

The authors would like to thank the people whose comments on early drafts helped refine this paper: Leslie Lamport, Marvin Theimer, Helen Wang, Dahlia Malkhi, Mike Schroeder, Lidong Zhou, and especially Chandu Thekkath, who was always available to answer our detailed questions about Petal. Finally, we thank the anonymous reviewers and our shepherd, Maurice Herlihy, for their many helpful comments and suggestions.

11. REFERENCES

- [1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th OSDI*, pages 1–14, Boston, MA, Dec. 2002.
- [2] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. 10th SOSP*, pages 79–86, Orcas Island, WA, Dec. 1985.
- [3] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, New Orleans, LA, Feb. 1999.
- [4] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, Dec. 2001.
- [5] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. 17th SRDS*, pages 43–50, West Lafayette, IN, Oct. 1998.
- [6] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), Apr. 1996.
- [7] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proc. 5th OSDI*, pages 147–163, Boston, MA, Dec. 2002.
- [8] R. Friedman and A. Vaysburd. Fast replicated state machines over partitionable networks. In *Proc. 16th SRDS*, pages 130–137, Durham, NC, Oct. 1997.
- [9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th SOSP*, pages 29–43, Bolton Landing, NY, Oct. 2003.
- [10] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, Sendai, Japan, June 1996.
- [11] J. Howell and J. Douceur. Replicated virtual machines. Technical report MSR-TR-2005-119, Microsoft Research, 2005.
- [12] J. Howell, J. R. Lorch, and J. Douceur. Correctness of Paxos with replica-set-specific views. Technical report MSR-TR-2004-45, Microsoft Research, 2004.
- [13] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [14] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.
- [15] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2003.
- [16] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th ASPLOS*, pages 84–92, Cambridge, MA, Oct. 1996.
- [17] N. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. 16th International Symposium on Distributed Computing*, pages 173–190, Toulouse, France, Oct. 2002.
- [18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. 6th OSDI*, pages 105–120, San Francisco, CA, Dec. 2004.
- [19] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *Proc. 2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 325–334, Florence, Italy, Jun. 2004.
- [20] B. M. Oki. Viewstamped replication for highly available distributed systems. Ph.D. thesis technical report MIT/LCS/TR-423, MIT, Aug. 1988.
- [21] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstractions to improve fault tolerance. In *Proc. 18th SOSP*, pages 15–28, Banff, Canada, Oct. 2001.
- [22] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.
- [23] C. A. Thekkath. *Personal communication*. 2005.
- [24] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th OSDI*, pages 91–104, San Francisco, CA, Dec. 2004.
- [25] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, pages 253–267, Bolton Landing, NY, Oct. 2003.
- [26] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. In *Proc. 1st NSDI*, pages 323–336, San Francisco, CA, Mar. 2004.