Combining Abstraction with Byzantine Fault-Tolerance

Rodrigo Rodrigues

May 24, 2001

This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. The author was partially supported by a Praxis XXI fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts, USA

Combining Abstraction with Byzantine Fault-Tolerance

by

Rodrigo Rodrigues

Abstract

This thesis describes a technique to build replicated services that combines Byzantine fault tolerance with work on abstract data types. Tolerating Byzantine faults is important because software errors are a major cause of outages and they can make faulty replicas behave arbitrarily. Abstraction hides implementation details to enable the reuse of existing service implementations and to improve the ability to mask software errors.

We improve resilience to software errors by enabling the recovery of faulty replicas using state stored in replicas with distinct implementations; using an opportunistic N-version programming technique that runs distinct, off-the-shelf implementations at each replica to reduce the probability of common mode failures; and periodically repairing each replica using an abstract view of the state stored by the correct replicas in the group, which improves tolerance to faults due to software aging.

We have built two replicated services that demonstrate the use of this technique. The first is an NFS service where each replica runs a different off-the-shelf file system implementation. The second is a replicated version of the Thor object-oriented database. In this case, the methodology enabled reuse of the existing database code, which is non-deterministic. These examples suggest that our technique can be used in practice: Our performance results show that the replicated systems perform comparably to their original, non-replicated versions. Furthermore, both implementations required only a modest amount of new code, which reduces the likelihood of introducing more errors and keeps the monetary cost of using our technique low.

Keywords: abstraction, abstraction function, abstract specification, Byzantine faults, fault tolerance, high availability, N-version programming, replication, software errors, wrapper.

Support: This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory. The author was partially supported by a Praxis XXI fellowship.

This report is a minor revision of the dissertation of the same title submitted to the Department of Electrical Engineering and Computer Science, in partial fulfillment of the requirements for the degree of Master of Science in that department. The thesis was supervised by Professor Barbara Liskov.

Para o meu avô Vítor Hugo

O poeta é um fingidor.
Finge tão completamente
Que chega a fingir que é dor
A dor que deveras sente.

Fernando Pessoa, Autopsicografia

To my grandfather Vítor Hugo

The poet is a pretender.
He pretends so absolutely
He manages to pretend a pain
From the real pain he feels.

Fernando Pessoa, Self-Psychography

Contents

1 Introduction **13**

1.1 Contributions 14

1.2 Thesis Outline 17

2 Practical Byzantine Fault Tolerance **19**

2.1 System Model and Assumptions 19

2.2 Algorithm Properties 20

2.3 Algorithm Overview 21

2.3.1 Processing Requests 22

2.3.2 View Changes 24

2.4 Proactive Recovery 24

2.5 State Transfer 25

2.6 Library Interface 27

3 The BASE Technique **29**

3.1 Methodology 29

3.2 The BASE Library 31

3.2.1 Interface 31

3.2.2 Implementation Techniques 35

4 Example I: File System **39**

4.1 Abstract Specification 39

4.2 Conformance Wrapper 40

4.3 State Conversions 42

4.4 Proactive Recovery 44

4.5 Discussion 45

5 Example II: Object-Oriented Database 47

5.1 System Overview 47

5.1.1 Object format and fetching 48

5.1.2 Transactions 49

5.1.3 The Modified Object Buffer 52

5.2 Abstract Specification 54

5.3 Conformance Wrapper 56

5.4 State Conversions 58

5.5 Discussion 59

5.5.1 Multiple Server Architecture 59

5.5.2 Proactive Recovery 61

5.5.3 Garbage Collector 61

6 Evaluation 63

6.1 Replicated File System 63

6.1.1 Overhead 64

6.1.2 Code Complexity 69

6.2 Replicated Object-Oriented Database 69

6.2.1 Overhead 69

6.2.2 Code Complexity 73

7 Related Work 75

7.1 Software Rejuvenation 75

7.2 N-Version Programming 76

8 Conclusions 79

8.1 Summary 79

8.2 Future Work 81

List of Figures

2-1 BFT algorithm in normal case operation 23
2-2 BFT view change algorithm 25
2-3 BFT interface and upcalls 27

3-1 BASE interface and upcalls 31
3-2 Overview of BASE interface 34

4-1 Software architecture 41
4-2 Example of the abstraction function 42
4-3 Inverse abstraction function pseudocode 43

5-1 Applications, frontends and object repositories in Thor 48
5-2 Server organization 53

6-1 Elapsed time, hot read-only traversals 71
6-2 Elapsed time, hot read-write traversals 71
6-3 Breakdown of T2b traversal 72
6-4 Breakdown of checkpointing 73

List of Tables

6.1 Andrew100: elapsed time in seconds 66
6.2 Andrew500: elapsed time in seconds 66
6.3 Andrew with proactive recovery: elapsed time in seconds. 67
6.4 Andrew: maximum time to complete a recovery in seconds. 67
6.5 Andrew100 heterogeneous: elapsed time in seconds 68

Acknowledgments

I was very fortunate to work with two of the most talented people I have ever met. Barbara Liskov and Miguel Castro earned all the respect and admiration I have for them. I sincerely thank them for all the guidance and support. This thesis would not be possible without their supervision.

All the people at the Programming Methodology Group were very supportive and contributed to a great work environment. I want to thank all the group members I had the pleasure to work with: Sarah Ahmed, Sameer Ajmani, Chandrasekhar Boyapati, Kyle Jamieson, Paul Johnson, Liuba Shrira, Michiharu Takemoto, Ziqiang Tang, Shan-Ming Woo, and Yan Zhang. Special thanks to Kincade Dunn for always taking care of us in such a spectacular way and for free therapy during spare times.

Many people in the Laboratory for Computer Science helped me with this work. Everyone in the Parallel and Distributed Operating Systems Group was very helpful in providing me with an infrastructure to run some of my experiments. I would like to give special thanks to Chuck Blake, Benjie Chen, Dorothy Curtis, Frank Dabek, Kevin Fu, Frans Kaashoek, Jinyang Li, David Mazières and Robert Morris.

I want to thank everyone at the Distributed Systems Group of INESC for all the guidance that led me to this point. I enjoyed very much the time I spent there, and it has also taught me a lot. I certainly miss the endless conversations at lunchtime where everyone said what they thought about subjects they knew nothing about. After I came here they have always welcomed me back to the group whenever I needed and they have also given me valuable feedback about my work.

Living in Boston for the last two years has been an enriching experience. I was fortunate to meet a variety of interesting people during my stay so far, and I hope to meet many more. I want to thank the lovely gang here in Boston for all the friendship and support.

I want to thank my wonderful family. I cannot imagine myself here without your support, dedication and love. Thanks to my father, sister, uncle, aunt, and Rosa for all the love they have given me. I hope I was able to give you some as well.

Last, but definitely not least, I want to thank all my friends in Lisbon who never forget me and make me always willing to return there. You have made coming here the most difficult decision in my life, and at the same time I could not have made this without your love. Thanks to you I will always know that home is where the heart is, and there is no need for you to worry: I am definitely going back.

Chapter 1

Introduction

Software errors are becoming increasingly common due to both the growth in size and complexity of software and the increasing need to move innovative products to the market as fast as possible. As the importance of computers in society increases, the need to make these programs more dependable also grows. Furthermore, there is an increasing number of malicious attacks that exploit software errors to gain control or deny access to systems that provide important services [19].

Service reliability can be achieved by replication. The idea is that even though some replicas may fail due to a hardware failure, software error, or malicious attack, the system as a whole can continue to provide service because the other replicas are still running.

Replicated systems are implemented by means of a replication algorithm that ensures the system as a whole behaves correctly even though some of the replicas are faulty. In the case of software errors and malicious attacks, faulty nodes can behave in arbitrarily bad ways. Such failures are called *Byzantine failures*. For example, a failed node may appear to be behaving properly, and yet at the same time be corrupting its state.

In [16, 17, 13], Castro and Liskov propose BFT, a replication algorithm for Byzantine fault-tolerance in asynchronous systems that offers good performance and strong correctness guarantees provided no more than $1/3$ of the replicas fail within a small window of vulnerability. Therefore, this algorithm allows systems to be highly available provided the replicas are not likely to fail at the same time.

However, BFT has two shortcomings that undermine its applicability.

1. It forces all replicas to agree on their entire state and update it in a deterministic way. This greatly reduces the applications that can use it:

- It precludes the use of nondeterministic implementations, which are very common. Useful nondeterministic operations include reading local clocks, using multi-threaded access to common data, or even using distinct implementations of memory allocation algorithms.
- It also precludes the use of distinct implementations at the replicas, yet using different implementations is a plausible way of avoiding a replicated system in which all replicas fail simultaneously due to a software error.

2. It does not make a clear separation between the application and the replica code. BFT forces the application to define its state as a contiguous memory region, and to notify the replication code whenever that state is about to be changed. This makes the reuse of existing implementations hard.

This thesis describes a replication technique that solves these problems by combining Byzantine fault tolerance [47] with work on data abstraction [38]. The next section describes our contributions in more detail.

1.1 Contributions

The replication technique presented in this thesis allows each replica to run distinct service implementations or to run implementations with non-deterministic behavior provided they share the same abstract behavior. This improves tolerance to software faults by reducing the probability of several replicas failing at the same time. Abstraction also improves fault tolerance by enabling the system to periodically repair the state of each replica using the abstract states stored by the others while hiding corrupt portions of their concrete states.

We propose a methodology to build replicated systems by reusing off-the-shelf implementations. It is based on the concepts of *abstract specification* and *abstraction function* from work on abstract data types [38]. We start by defining an *abstract specification*; the specification describes an *abstract state* for the service and how that state is manipulated by each service operation. Then, for each distinct service implementation we implement a *conformance wrapper* that maps from the concrete behavior of that implementation to the required abstract behavior. We also implement the abstraction function and one of its inverses to map from the concrete state of the implementation to the abstract state and vice versa.

The methodology offers several important advantages.

- **Reuse of existing code.** BFT implements state machine replication [54, 33]. The state machine approach is useful because it allows replication of services that perform arbitrary computations, but it requires determinism: all replicas must produce the same sequence of results when they process the same sequence of operations. The use of a conformance wrapper that hides nondeterminism in the behavior of the applications and the fact that replicas have to agree on an abstract state instead of a concrete state enables the reuse of existing implementations without modifications.

- **Software rejuvenation through proactive recovery.** It has been observed [31] that there is a correlation between the length of time software runs and the probability that it fails. Software rejuvenation [31] is a fault-tolerance technique inspired by this observation. It periodically stops an application, dumps its state to disk, and restarts the application from the saved state; the hope is to restart soon enough that the saved state is still correct.

  Our methodology makes it possible to use proactive recovery as a form of software rejuvenation. Replicas are recovered periodically even if there is no reason to suspect they are faulty. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas. This improves on previous techniques by combining rejuvenation with Byzantine-fault-tolerant replication:

  - rejuvenation can be performed frequently because the service remains available during rejuvenation;
  - it works even if the state of the recovering replica is corrupt because it can obtain a correct copy of the state from the group;
  - it works even if all replicas have corrupt portions in their concrete states provided these are hidden by abstraction; and
  - recoveries are staggered such that individual replicas are highly unlikely to fail simultaneously because at any point they have been running for different lengths of time.

- **Efficient and Opportunistic N-version programming.** Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas (as would be the case when there is a deterministic software bug). N-version programming [20]

exploits design diversity to reduce the probability of correlated failures, but it has several problems [25] :

- First, it does not provide an efficient state comparison and transfer mechanism. Previous work in N-version programming either ignores the issue of state checking and transfer or addresses it with ad-hoc, impractical solutions.

  We provide an answer to this problem by comparing the entire state of the replicas during periodic checkpoints and after proactively recovering a replica. Furthermore, our state comparison and transfer mechanism is efficient because it is based on maintaining a tree with digests of the components of the state. This allows the state checking mechanism to quickly determine which components of the state are incorrect, and the check requires a limited amount of data transfer. It also enables us to fetch only the incorrect subset of the state.

  The state checking and transfer is done periodically without any need of additional specialized code. This reduces the complexity of using our mechanism, avoiding the likelihood of introducing additional errors. Doing checks automatically is also a good way to avoid errors that might result from some correlation between the code and the checks.

  Also, since we are comparing abstract state we allow greater diversity (and therefore reduce the likelihood of correlated errors) because the different implementations need not implement identical specifications. Instead they only need to implement similar specifications that can be mapped by conformance wrappers into the same abstract behavior and they also only need to agree on the same abstract state via the abstraction function.

- Another problem is that N-version programming does not does not address the issue of how to reintegrate a replica that has failed with the working system without interrupting the service. Our methodology solves this problem since the use of abstraction permits us to know the correct state of the system at any point. Therefore, we provide a framework that allows the periodic repair of faulty replicas using proactive recovery and the abstract state transfer mechanism.

- The other criticism that has been made of N-version programming is that it increases development and maintenance costs by a factor of N or more, and adds unacceptable time delays to the implementation.

  Our methodology enables an opportunistic form of N-version programming by allowing

us to take advantage of distinct, off-the-shelf implementations of common services. This approach overcomes the defects mentioned above: it eliminates the high development and maintenance costs of N-version programming, and also the long time-to-market.

Opportunistic N-version programming is a viable option for many common services, e.g., relational databases, HTTP daemons, file systems, and operating systems. In all these cases, competition has led to four or more distinct implementations that were developed and maintained separately but have similar (although not identical) functionality. Furthermore, the technique is made easier by the existence of standard protocols that attempt to provide identical interfaces to different implementations, e.g., ODBC [23] and NFS [2]. We can also leverage the large effort towards standardizing data representations using XML.

Our replication technique is implemented by the BASE library, which extends the BFT library (BASE is an acronym for BFT with Abstract State Exchange). BASE overcomes BFT's defects that were mentioned above. This allows BASE to support our methodology in a simple and efficient way.

This thesis also presents the design and implementation of two replicated services using our replication technique: a replicated file service where replicas run different operating systems and file systems, and an object-oriented database server that uses a single but non-deterministic implementation for all the replicas.

For this methodology to be successful, the conformance wrapper and the state conversion functions must be simple to reduce the likelihood of introducing more errors and introduce a low overhead. We provide an evaluation of our example applications using these metrics. Our results show that our technique can be used in practice. The replicated versions for these systems require a small amount of new code and perform comparably to the original systems.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the replication algorithm that we extended to support the use of abstraction. In Chapter 3 we present our replication technique: its methodology and the library that supports it, BASE. Chapters 4 and 5 describe the two example services that were developed using our replication technique: an NFS file service and an object-oriented database server, respectively. For each application we describe the implementation of the

various steps of our methodology in these two chapters. We present an experimental evaluation for the two replicated services in Chapter 6. These results try to show that both services met the requirements that we need for the technique to be successful: it must introduce a low overhead and require a small amount of new code. We compare our approach to previous work in Chapter 7. Finally, Chapter 8 concludes the thesis and mentions the areas for future work.

Chapter 2

Practical Byzantine Fault Tolerance

This chapter provides an overview of a practical Byzantine fault tolerance replication algorithm [16, 17] and BFT, the library that implements it. We discuss only those aspects of the algorithm and library that are relevant to this thesis; for a complete description, see [13].

The algorithm enables proactive recovery of replicas, which allows it to tolerate any number of faults over the lifetime of the system, provided less than 1/3 of the replicas become faulty within a small window of vulnerability. The algorithm uses mainly symmetric cryptography, which allows it to perform well so that it can be used in practice to implement real services.

We begin by describing the system model and assumptions. Section 2.2 describes the problem solved by the algorithm and its underlying assumptions. Section 2.3 gives an overview of the algorithm. Sections 2.4 and 2.5 discuss two important aspects of the algorithm: the proactive recovery and the state transfer mechanisms, respectively. Finally, Section 2.6 presents the interface for the BFT library.

2.1 System Model and Assumptions

The algorithm assumes an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order. It assumes a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the restrictions mentioned below. An adversary is allowed to coordinate faulty nodes, delay communication, inject messages into the network, or delay correct nodes in order to cause the most damage to the replicated service. The adversary cannot delay correct nodes indefinitely, though.

Cryptographic techniques are employed to establish session keys, authenticate messages, and

produce digests. It is assumed that the adversary (and the faulty nodes it controls) is computationally bound so that it is unable to subvert these cryptographic techniques.

These are all the assumptions that are required to provide safety if less than 1/3 of the replicas become faulty during the lifetime of the system. To tolerate more faults a few additional assumptions are needed: a secure cryptographic co-processor must be employed to sign and decrypt messages without exposing the replica's private key, so that it is possible to mutually authenticate a faulty replica that recovers to the other replicas; a read-only memory must store a correct copy of the service code to be used after a recovery; and a reliable mechanism such as a watchdog timer is needed to trigger periodic recoveries.

With these assumptions, the system is able to perform frequent recoveries without relying on system administrators to assist in the process.

2.2 Algorithm Properties

The algorithm is a form of state machine replication [54, 33]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. The algorithm can be used to implement any replicated service with a state and some operations. The operations are not restricted to simple reads and writes; they can perform arbitrary computations.

The service is implemented by a set of replicas R and each replica is identified using an integer in $\{0, ..., |R| - 1\}$. Each replica maintains a copy of the service state and implements the service operations. For simplicity, it is assumed that $|R| = 3f + 1$ where f is the maximum number of replicas that may be faulty.

Like all state machine replication techniques, this algorithm requires each replica to keep a local copy of the service state. All replicas must start in the same internal state, and they also must be deterministic, in the sense that the execution of an operation in a given state and with a given set of arguments must always produce the same result and lead to the same state following that execution.

This algorithm ensures *safety* for an execution provided at most f replicas become faulty within a window of vulnerability of size T_v. Safety means that the replicated service satisfies linearizability [29]: it behaves like a centralized implementation that executes operations atomically one at a time. A safety proof for a simplified version of the algorithm using the I/O automata formalism [39] is sketched in [15].

The algorithm also guarantees liveness: non-faulty clients eventually receive replies to their

requests provided (1) at most f replicas become faulty within the window of vulnerability T_v; and (2) denial-of-service attacks do not last forever, i.e., there is some unknown point in the execution after which all messages are delivered (possibly after being retransmitted) within some constant time d, or all non-faulty clients have received replies to their requests. A liveness proof for the same simplified version of the algorithm is also provided in [51].

2.3 Algorithm Overview

The algorithm works roughly as follows. Clients send requests to execute operations to the replicas and all non-faulty replicas execute the same operations in the same order. Since replicas are deterministic and start in the same state, all non-faulty replicas send replies with identical results for each operation. The client waits for f+1 replies from different replicas with the same result. Since at least one of these replicas is not faulty, this is the correct result of the operation.

The hard problem is guaranteeing that all non-faulty replicas agree on a total order for the execution of requests despite failures. A primary-backup mechanism is used to achieve this. In such a mechanism, replicas move through a succession of configurations called views. In a view one replica is the primary and the others are backups. The primary of a view is chosen to be replica p such that $p = v \; mod \; |R|$ where v is the view number and views are numbered consecutively.

The primary picks the ordering for execution of operations requested by clients. It does this by assigning a sequence number to each request. But the primary may be faulty. Therefore, the backups trigger view changes when it appears that the primary has failed to propose a sequence number that would allow the request to be executed.

To tolerate Byzantine faults, every step taken by a node in this system is based on obtaining a certificate. A certificate is a set of messages certifying some statement is correct and coming from different replicas. An example of a statement is: ”the result of the operation requested by a client is *r*”.

The size of the set of messages in a certificate is either $f + 1$ or $2f + 1$, depending on the type of statement and step being taken. The correctness of the system depends on a certificate never containing more than f messages sent by faulty replicas. A certificate of size $f + 1$ is sufficient to prove that the statement is correct because it contains at least one message from a non-faulty replica. A certificate of size $2f + 1$ ensures that it will also be possible to convince other replicas of the validity of the statement even when f replicas are faulty.

Other Byzantine fault-tolerance algorithms [48, 32, 16] rely on the power of digital signatures to authenticate messages and build certificates. This algorithm uses message authentication codes (MACs) [9] to authenticate all messages in the protocol. A MAC is a small bit string that is a function of the message and a key that is shared only between the sender and the receiver. The sender appends this to the protocol messages so that the receiver can check the authenticity of the message by computing the MAC in the same way and comparing it to the one appended in the message.

The use of MACs substantially improves the performance of the algorithm — MACs, unlike digital signatures, use symmetric cryptography instead of public-key cryptography — but also makes it more complicated: the receiver may be unable to convince a third party that a message is authentic, since the third party must not know the key that was used to generate its MAC.

Since we wish to tolerate more than f faults throughout the execution, we must periodically change the session keys that are used between each pair of replicas. When this happens, the replica must discard all messages in its log that are not part of a complete certificate and it will reject any messages it receives in the future that are authenticated with old keys. This ensures that correct nodes only accept certificates with equally fresh messages, i.e., messages authenticated with keys created in the same refreshment phase, so that they do not collect messages for a certificate over a period of time that is so long that they end up with more than f messages from faulty replicas.

The messages used in the key exchange period are digitally signed using a secure co-processor that stores the replica's private key. This prevents an attacker from compromising the replica's private key, even if she gains control of the machine.

2.3.1 Processing Requests

When the primary receives a request, it uses a three-phase protocol to atomically multicast requests to the replicas. The three phases are pre-prepare, prepare, and commit. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views.

Figure 2-1 shows the operation of the algorithm in the normal case of no primary faults. In this example replica 0 is the primary and replica 3 is faulty. The client begins by sending a request to the primary, which multicasts it to all replicas in a pre-prepare message. This message proposes a sequence number for the request, and if the remaining replicas agree with this sequence number

they multicast a prepare message. When each replica collects a certificate with $2f+1$ matching prepare messages from different replicas (possibly including its own), it multicasts a commit message. When a replica has accepted $2f + 1$ commit messages that match the pre-prepare for the request, it executes the request, causing its state to be updated and producing a reply. This reply is sent directly to the client, who waits for $f + 1$ replies from different replicas with the same result.

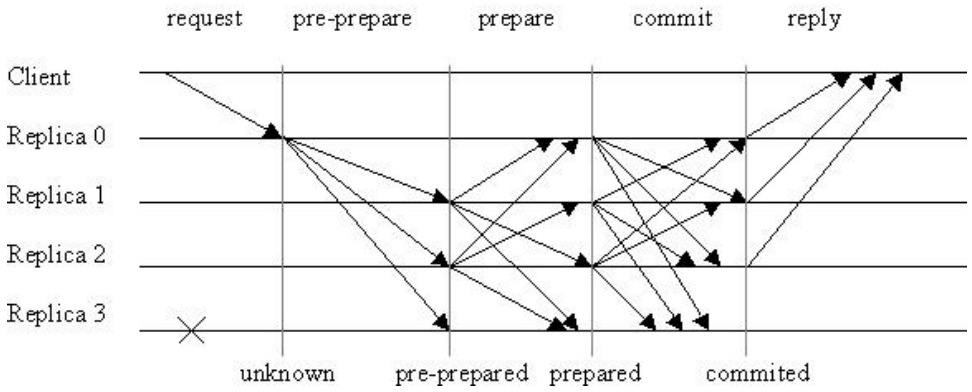

Figure 2-1: BFT algorithm in normal case operation

Each replica stores the service state, a *log* containing information about requests, and an integer denoting the replica's current view. The log records information about the request associated with each sequence number, including its status; the possibilities are: *unknown* (the initial status), *pre-prepared*, *prepared*, and *committed*. Figure 2-1 also shows the evolution of the request status as the protocol progresses.

Replicas can discard entries from the log once the corresponding requests have been executed by at least $f+1$ non-faulty replicas, a condition required to ensure that request will be known after a view change. The algorithm reduces the cost by determining the condition only when a request with a sequence number divisible by some constant K (e.g., $K = 128$) is executed. The state produced by the execution of such requests is called a *checkpoint*. When a replica produces a checkpoint, it multicasts to other replicas a checkpoint message containing a digest of its state d, and the sequence number of the last request whose execution is reflected in the state, n. Then, it waits until it has a certificate with $2f + 1$ valid checkpoint messages for the same sequence number n and with the same state digest d sent by different replicas. At this point the checkpoint is known to be stable and the replica garbage collects all entries in its log with sequence numbers less than or equal to n; it also discards earlier checkpoints.

Creating checkpoints by making full copies of the state would be too expensive. Instead, the library uses copy-on-write such that checkpoints only contain the differences relative to the current

state. The data structures used in efficiently computing the digest d of the state are presented in Section 2.5.

2.3.2 View Changes

The view change protocol provides liveness by allowing the system to make progress when the current primary fails. The protocol must preserve safety: it must ensure that non-faulty replicas agree on the sequence numbers of committed requests across views. In addition, to provide liveness it must ensure that non-faulty replicas stay in the same view long enough for the system to make progress, even in the face of a denial-of-service attack.

View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is waiting for a request if it received a valid request and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v, the backup starts a view change to move the system to view $v + 1$. It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a view-change message to all replicas. Figure 2-2 illustrates this situation.

The new primary p for view $v + 1$ collects a certificate with $2f + 1$ valid view-change messages for view $v + 1$ signed by different replicas. After obtaining the new-view certificate and making necessary updates to its log, p multicasts a new-view message to all other replicas, and enters view $v + 1$: at this point it is able to accept messages for view $v + 1$. A backup accepts a new-view message for $v + 1$ if it is properly signed, if it contains a valid new-view certificate, and if the message sequence number assignments do not conflict with requests that committed in previous views. The backup then enters view $v + 1$, and becomes ready to accept messages for this new view.

This description hides a lot of the complexity of the view change algorithm, namely the difficulties that arise from using MACs instead of digital signatures. A detailed description of the protocol can be found in [13, 17].

2.4 Proactive Recovery

Proactive recovery makes faulty replicas behave correctly again, allowing the system to tolerate more than f faults over its lifetime. Recovery is periodic and independent of any failure detection

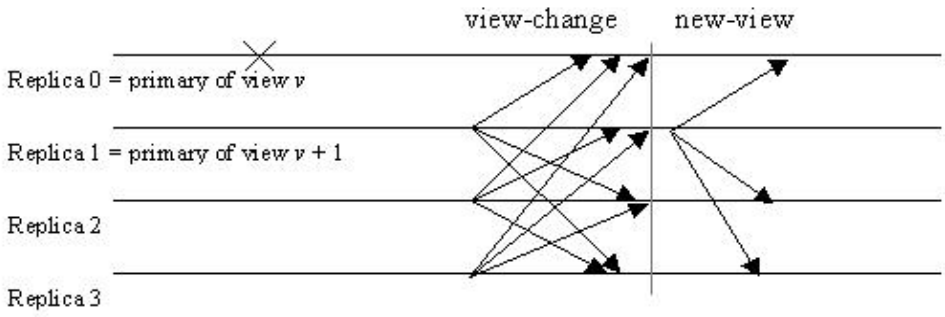

Figure 2-2: BFT view change algorithm

mechanism, since faulty replicas may appear to behave properly even when broken. After recovery, each replica restarts from a correct, up-to-date checkpoint of the state that is obtained from the other replicas.

Recoveries are staggered so that fewer than $1/3$ of the replicas recover at the same time. This allows the other replicas to continue processing client requests during the recovery. Additionally, it should reduce the likelihood of simultaneous failures due to aging problems because at any instant fewer than $1/3$ of the replicas have been running for the same period of time.

Recoveries are triggered by a watchdog timer. When a replica is recovered, it reboots with correct code (obtained from a read-only memory) after saving the replication protocol state and the service state to disk. Then the replica is restarted, and the state that was previously saved is read from disk.

The recovering replica must then change its session keys used to authenticate messages. This is important because these keys may be known to the attacker if the recovering replica was faulty. By changing these keys, we prevent the attacker from impersonating any client or another replica after recovery, and bound the sequence number of messages forged by the attacker that may be accepted by the other replicas.

Next, the library uses the hierarchical state transfer mechanism that we describe in Section 2.5 to efficiently compare the value of the state that was read from disk with the state values stored by the other replicas, and fetch only the part of the state that is out-of-date or corrupt.

2.5 State Transfer

In BFT, state is defined as a fixed-size, contiguous region of memory. When a replica is initialized, it must inform the library of the starting location and size of that region of memory that defines the

service state. All replicas must agree on the initial value and the transformations operated by each request on the contents of that region.

As we mentioned, checkpoints are snapshots of the service state of each replica that are taken periodically. A replica may learn about a stable checkpoint beyond the highest possible sequence number in its log by receiving checkpoint messages or as the result of a view change. In this case, it uses the state transfer mechanism to fetch modifications to the service state that it is missing.

It is important for the state transfer mechanism to be efficient because it is used to bring a replica up to date during recovery, and we perform proactive recoveries frequently. The key issues to achieving efficiency are reducing the amount of information transferred and reducing the burden imposed on replicas.

To achieve this, we use a data structure called a *partition tree* that forms a hierarchical state partition. The root partition corresponds to the entire service state and each non-leaf partition is divided into s equal-sized, contiguous sub-partitions. We call leaf partitions pages and interior partitions meta-data. The pages correspond to a fixed-size, contiguous subset of the replica's state.

Each replica maintains one logical copy of the partition tree for each checkpoint. The copy is created when the checkpoint is taken and it is discarded when a later checkpoint becomes stable. The tree for a checkpoint stores a tuple $< lm, d >$ for each meta-data partition and a tuple $< lm, d, p >$ for each page. Here, lm is the sequence number of the checkpoint at the end of the last checkpoint interval where the partition was modified, d is the digest of the partition, and p is the value of the page.

The digests are computed efficiently as follows. For a given page, d is obtained by applying the MD5 hash function [49] to the string obtained by concatenating the index of the page within the state, its value of lm, and p. For meta-data, d is obtained by applying MD5 to the string obtained by concatenating the index of the partition within its level, its value of lm, and the sum modulo a large integer of the digests of its sub-partitions. Thus, we apply AdHash [8] at each meta-data level. This construction has the advantage that the digests for a checkpoint can be obtained efficiently by updating the digests from the previous checkpoint incrementally.

The copies of the partition tree are logical because copy-on-write is used so that only copies of the tuples modified since the checkpoint was taken are stored. This reduces the space and time overheads for maintaining these checkpoints significantly. The details of how this was implemented will be mentioned in Section 3.2.2, when we discuss some of techniques used in the implementation of our system.

When a replica is fetching state, it recurses down the hierarchy of meta-data to determine which partitions are out of date by comparing its own digest and sequence number of last modification with the ones it receives in response to *fetch* messages. When it reaches the leaves of the hierarchy (which are the pages that make up a partition of the state), it sends a fetch request for each page that it found to be corrupt or out of date.

To minimize the amount of data transferred and the overhead that is imposed on the replicas, when the recovering replica sends a fetch message, one replica is designated as the replier. This replica responds with the full page while the others only send the digest and last modification sequence number for the page. These are used to confirm that new value for the page is correct and up-to-date.

2.6 Library Interface

The BFT library implements the algorithm that has been described before. Its basic interface is depicted in Figure 2-3.

```
Client call:
int invoke(Byz_req *req, Byz_rep *rep, bool read_only);

Execution upcall:
int execute(Byz_req *req, Byz_rep *rep, int client, bool read-only);

Checkpointing:
void modify(char *mem, int size);
```

Figure 2-3: BFT interface and upcalls

The `invoke` procedure is called by the client to invoke an operation on the replicated service. This procedure carries out the client side of the replication protocol and returns the result when enough replicas have responded.

When the library needs to execute an operation at a replica, it makes an upcall to an `execute` procedure that carries out the operation as specified for the service. The arguments to this procedure include a buffer with the requested operation and its arguments, a buffer to fill with the operation result, the identifier of the client who requested the operation, and a boolean flag indicating whether a request was processed with the read-only optimization. The service code can use the client identifier to perform access control, and the read-only flag to reject operations that modify the state but were flagged read-only by faulty clients.

Each time the `execute` upcall is about to modify a part of the state it is required to invoke a `modify` procedure, which is supplied by the library, passing the starting address and size of the memory region that is about to be modified as arguments. This is used to implement copy-on-write to create checkpoints incrementally: the library keeps copies of the pages that were modified until the corresponding checkpoint can be discarded.

Chapter 3

The BASE Technique

This chapter provides an overview of our replication technique. It starts by describing the methodology that we use to build a replicated system from existing service implementations. It ends with a description of the BASE library. This technique was first proposed in [18] and a more comprehensive description can be found in [50].

3.1 Methodology

The goal is to build a replicated system by reusing a set of off-the-shelf implementations, $I_1, ..., I_n$, of some service. Ideally, we would like n to equal the number of replicas so that each replica can run a different implementation to reduce the probability of simultaneous failures. But the technique is useful even with a single implementation.

Although off-the-shelf implementations of the same service offer roughly the same functionality, they behave differently: they implement different specifications, $S_1, ..., S_n$, using different representations of the service state. Even the behavior of different replicas that run the same implementation may be different when the specification they implement is not strong enough to ensure deterministic behavior. For instance, the specification of the NFS protocol [2] allows implementations to choose the values of file handles arbitrarily.

BASE, like any form of state machine replication, requires determinism: replicas must produce the same sequence of results when they execute the same sequence of operations. We achieve determinism by defining a *common abstract specification*, S, for the service that is strong enough to ensure deterministic behavior. This specification defines the abstract state, an initial state value, and the behavior of each service operation.

The specification is defined without knowledge of the internals of each implementation. It is sufficient to treat implementations as black boxes, which is important to enable the use of existing implementations. Additionally, the abstract state captures only what is visible to the client rather than mimicking what is common in the concrete states of the different implementations. This simplifies the abstract state and improves the effectiveness of our software rejuvenation technique.

The next step is to implement *conformance wrappers*, $C_1, ..., C_n$, for each of $I_1, ..., I_n$. The conformance wrappers implement the common specification S. The implementation of each wrapper C_i is a veneer that invokes the operations offered by I_i to implement the operations in S; in implementing these operations the conformance wrapper makes use of a *conformance rep* that stores whatever additional information is needed to allow the translation from the concrete behavior of the implementation to the abstract behavior. The conformance wrapper also implements some additional methods that allow a replica to be shutdown and then restarted without loss of information.

The final step is to implement the *abstraction function* and one of its inverses. These functions allow state transfer among the replicas. State transfer is used to repair faulty replicas, and also to bring slow replicas up-to-date when messages they are missing have been garbage collected. For state transfer to work, replicas must agree on the value of the state of the service after executing a sequence of operations; they will not agree on the value of the concrete state but our methodology ensures that they will agree on the value of the abstract state. The abstraction function is used to convert the concrete state stored by a replica into the abstract state, which is transferred to another replica. The receiving replica uses the inverse function to convert the abstract state into its own concrete state representation.

The BFT library requires all replicas to agree on the value of the concrete state. This effectively rules out using different implementations or even using the same off-the-shelf implementation if it is non-deterministic. In BASE, replicas agree on the value of the abstract state, not the concrete state. Therefore, the state transfer mechanism transfers abstract state values.

It is not sufficient to have a mathematical model for the abstract state; it is necessary to define a precise encoding that can be used to transfer state efficiently. For this purpose, we impose that the abstract state must be defined as an array of objects. The array has a fixed maximum size, but the objects it contains can vary in size. We also require the abstraction function and its inverse to be implemented at the granularity of objects, instead of having a single map for the whole state, which enables efficient state transfer.

3.2 The BASE Library

In this section we present the BASE library. This library extends BFT with the features necessary to provide the methodology. We begin by presenting its interface and then discuss some implementation techniques.

3.2.1 Interface

The BASE library provides support for the methodology described in the previous section. Figure 3-1 presents the library's interface.

```
Client call:
int invoke(Byz_req *req, Byz_rep *rep,
           bool read_only);

Execution upcall:
int execute(Byz_req *req, Byz_rep *rep, int client,
            Byz_buffer *non-det, bool read-only);

State conversion upcalls:
 - Abstraction function:
int get_obj(int i, char** obj);

 - Inverse abstraction function:
void put_objs(int n-objs, char **objs, int *indices, int *sizes);

Checkpointing:
void modify(int n-objs, int* objs);

Non-determinism upcalls:
int propose_value(Seqno seqno, Byz_buffer *req, Byz_buffer *non-det);

int check_value(Seqno seqno, Byz_buffer *req, Byz_buffer *non-det);

Recovery upcalls:
void shutdown_proc(FILE *out);

void restart_proc(FILE *in);
```

Figure 3-1: BASE interface and upcalls

The `invoke` and `execute` procedures have the same interface and semantics as in the BFT library (see Section 2.6), except for an additional parameter to `execute` that will be explained shortly.

As mentioned earlier, to implement checkpointing and state transfer efficiently, we require that the abstract state be encoded as an array of objects, where the array has a fixed maximum size but

the objects can have variable size. This representation allows state transfer to be done on just those objects that are out of date or corrupted.

To implement state transfer, each replica must provide the library with two upcalls, which implement the *abstraction function* and one of its inverses. These cannot be correspondences between the entire concrete and abstract states, since the overhead of computing the entire mapping would be prohibitively large. Instead, we require the application to implement the abstraction function at the granularity of an object in the abstract state array. `get_obj` receives an object index i, allocates a buffer, obtains the value of the abstract object with index i, and places that value in the buffer. It returns the size for that object and a pointer to the buffer.

The inverse abstraction function causes the application to update its concrete state using the new value for the abstract state passed as argument. Again, this function should not update the entire state, but it also cannot be defined to process just one object, since this may lead correct but slow replicas to incoherent intermediate states, where certain invariants concerning the abstract state do not hold.

We illustrate this problem with an example. Suppose we model a file system where an object in the abstract state can be a file or a directory. If a correct but slow replica failed to create a directory and a file inside that directory, the library must invoke the inverse abstraction function in order to create these two objects in the concrete state. If it invoked this function twice, passing only the file object in the first call and its parent directory in the second call, it would be impossible for the implementation of the inverse abstraction function to update the concrete state after the first call, since it would have no information on where to create the file. The reverse order also produces an incoherent intermediate state, since we end up with a directory that has an entry whose type and contents are not described in the abstract state. Also, even if a particular ordering might work, there is no simple way to tell the library which order to use when invoking the inverse abstraction function with multiple objects.

To avoid this situation, `put_objs` receives a vector of objects with the corresponding sizes and indices in the abstract state array. The library guarantees that this upcall is invoked with an argument that brings the abstract state of the replica to a consistent value (i.e., the value of a valid checkpoint).

Similarly to what happened in the BFT library, each time the `execute` upcall is about to modify an object in the abstract state it must invoke the `modify` procedure in order to implement copy-on-write to create checkpoints incrementally. The interface of this procedure is different from BFT, since we are now concerned about changes in the abstract state, and not the region of the

concrete state that has been modified. Therefore, the `modify` procedure supplied by the BASE library takes as arguments an array with the indices in the abstract state array of the objects that are going to be modified and the size of that array.

BASE implements a form of state machine replication that requires replicas to behave deterministically. Our methodology uses abstraction to hide most of the non-determinism in the implementations it reuses. However, many services involve forms of non-determinism that cannot be hidden by abstraction. For instance, in the case of the NFS service, the time-last-modified for each file is set by reading the server's local clock. If this were done independently at each replica, the states of the replicas would diverge.

Instead, we allow the primary replica to propose values for all non-deterministic choices by providing the `propose_value` upcall, which is only invoked at the primary. The call receives the client request and the sequence number for that request; it selects a non-deterministic value and puts it in `non-det`. This value is going to be supplied as an argument of the `execute` upcall to all replicas.

The protocol implemented by the BASE library prevents a faulty primary from causing replica state to diverge by sending different values to different backups. However, a faulty primary might send the same, incorrect value to all backups, subverting the system's desired behavior. The solution to this problem is to have each replica implement a `check_value` function that validates the choice of non-deterministic values that was made by the primary. If $1/3$ or more non-faulty replicas reject a value proposed by a faulty primary, the request will not be executed and the view change mechanism will cause the primary to be replaced soon after.

As explained in Section 2.4, proactive recovery periodically restarts each replica from a correct, up-to-date checkpoint of the state (now, the commonly perceived abstract state) that is obtained from the other replicas. During the replica shutdown, it must save the replication protocol state and the service state to disk. In the BFT library, saving the service state to disk could be done by the library code, since it had access to the concrete state of the service. In BASE, the state must be accessed using upcalls that are implemented by the service.

Saving the entire abstract state to disk would be very expensive, especially if the concrete state is already stored in disk, and we can reconstruct the abstract state from the concrete state just by reconstructing a small additional amount data (typically the information stored in the conformance rep). Therefore, we allow the service state to be saved by an upcall to the `shutdown` method provided by the conformance wrapper; this method is responsible for saving enough information to

the file that is received as a parameter, so that it is possible to later reconstruct the abstract state.

In some cases, the information in the conformance rep is volatile, in the sense that it is no longer valid when the replica restarts. In this case we must save some information that is persistent across reboots, and that allows the conformance rep to be reconstructed from it after a reboot.

When the replica is restarted, the `restart` procedure of the conformance wrapper is invoked to reconstruct the abstract state. This is done by reading the information that was previously stored to disk from the file that is received as a parameter. The replica reconstructs the conformance rep and the concrete state from the information in this file, which enables it to compute the abstract state from the concrete state.

Next, the library uses the hierarchical state transfer mechanism to compare the value of the abstract state of the replica with the abstract state values stored by the other replicas. This is efficient: the replica uses cryptographic hashes stored in the state partition tree to determine which abstract objects are out-of-date or corrupt and it only fetches the value of these objects.

The object values fetched by the replica could be supplied to `put_objs` to update the concrete state, but the concrete state might still be corrupt. For example, an implementation may have a memory leak and simply calling `put_objs` will not free unreferenced memory. In fact, implementations will not typically offer an interface that can be used to fix all corrupt data structures in their concrete state. Therefore, it is better to restart the implementation from a clean initial concrete state and use the abstract state to bring it up-to-date.

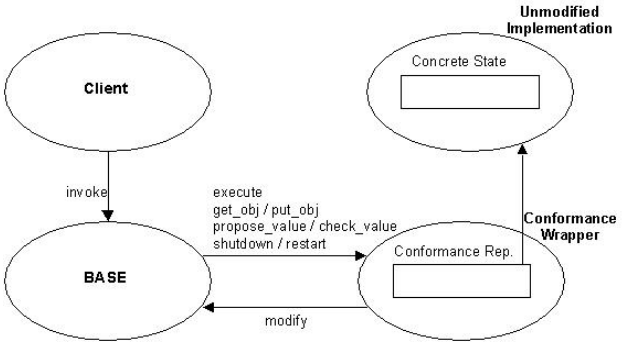

Figure 3-2: Overview of BASE interface

A global view of all BASE functions and upcalls that are invoked is shown in Figure 3-2.

3.2.2 Implementation Techniques

In this section we discuss some techniques that were used in implementing BASE. These describe the main extensions to the BFT library that were made to support the definition of an abstract state.

Checkpoint Management

In Section 2.5, we described a data structure, the partition tree, that allows BFT to efficiently (1) keep logical checkpoints that are based on the current state and old snapshots of pages that have changed; (2) compute an incremental digest of the entire state, which is appended to checkpoint messages that are sent after executing every K requests; and (3) determine which pages must be fetched when the state of a replica is corrupt or out-of-date.

Recall that in BFT each replica maintains one logical copy of the partition tree for each checkpoint that is not yet stable. The tree for a checkpoint stores a tuple $< lm, d >$ for each meta-data partition and a tuple $< lm, d, p >$ for each page, where lm is the sequence number of the checkpoint at the end of the last checkpoint interval where the partition was modified, d is the digest of the partition, and p is the value of the page.

This is implemented by having the following items in virtual memory:

1. a copy of the current state (all pages) — this is actually shared between the application and the replication library.

2. the digests in the partition tree computed in the last checkpoint

3. copies of pages and digests stored during an older checkpoint, provided that these pages or digest have been modified after that checkpoint.

In BASE, things are a bit different since there is a clear distinction between the application state (the concrete state) and the state that is managed by the library (the abstract state). Furthermore, we now represent the state as an array of objects, instead of a contiguous chunk of memory that is partitioned in pages.

Therefore, we need to change the partition tree, so that the leaves now store objects instead of pages. So the new tuple stored in the partition tree leaves is $< lm, p, d, sz, o >$, where sz is the size of the object and *o* is the abstract object.

We still expect to keep one logical copy of the partition tree for each checkpoint that is not yet stable. But this time, we only keep items (2) and (3) in the list above in virtual memory, i.e., we do

not expect to keep a copy of the entire current abstract state in memory, but instead we only ask for copies of the abstract objects when we need them.

This way, we expect to free a large amount of virtual memory for the unmodified server process that is invoked by the conformance wrapper. In BFT this was not a major problem since the state of the library and the application that implemented the service were shared.

During normal case operation (i.e. if we exclude state transfers) the abstraction function will be invoked on two kinds of occasions: during a checkpoint, to compute the new value of the object's digest, if it was changed during the last checkpoint interval; and before an object is modified for the first time in a checkpoint interval, so that we can store a logical copy of the state during the previous checkpoint (we only store copies of the abstract objects that change).

Normally, this will result in two consecutive calls of the abstraction function (first to compute the digest during the checkpoint, and next when the object is subsequently modified). The abstract object returned by these two calls will be the same.

We could minimize the overhead introduced by the abstraction function by keeping the abstract object in memory between these calls. But the interval between these two calls may be very large, and we would end up with almost the entirety of the abstract state in virtual memory.

Still, we can expect a certain locality in object access, and therefore we keep a small LRU cache with objects that were computed by the abstraction function during a checkpoint. These may be used to avoid another call to the abstraction function if the object is modified in the near future.

State Transfer

State transfer is now done at the granularity of objects instead of pages: the library detects which objects are out of date or corrupt and fetches them, in an analogous way that it detected and fetched out of date or corrupted pages in BFT. This raises a problem when we are trying to fetch an object with a size larger than the maximum message size of the underlying transport mechanism (in our case, UDP). To overcome this problem we must perform fragmentation of data messages (replies to fetches). We do this by introducing a constant that defines the maximum fragment size. When the recovering replica issues a fetch request, it must state which fragment of the object it is asking for. In a data message, the replier must include the fragment number it is sending and the total size of the object (so the recovering replica may know whether to send more fetches for the remaining fragments of the same object).

A problem arises from the fact that we designate a replier to the fetch message (the remaining

replicas only send digests), and a malicious replier might send an object size that is excessively large, causing the recovering replica to waste time sending various fetches for the same object. This problem is solved by a mechanism that periodically changes the replier of the fetch request. When the recovering replica notices a change in the object size it must restart the object fetch from the beginning.

Fetches of large objects may be problematic, since we pay the penalty of sending several fetches for the various fragments, even if only a small part of the object is incorrect. This problem may be solved by changing the definition of the abstract state, such that it generates smaller objects. An example of such an alteration will be shown in Section 4.5. In Section 8.2 we propose a modification in the BASE library that overcomes this problem without the need to change the abstract state specification.

Recoveries

After a replica restarts, the BASE library restores the state of the protocol using information that was saved during shutdown, like the value for its own partition tree, and then invokes the `restart` upcall of the conformance wrapper in order to reconstruct the abstract state from the concrete state that is in disk plus some extra information that was saved by the conformance wrapper using the `shutdown` method.

After that, the recovering replica must check its own abstract state and fetch those objects that are out-of-date or corrupt. To do this, the replica compares the digests in its own partition tree with the partition digests that it fetches from other replicas, assuming that the digest for the objects match the values that the library stored during shutdown. Then, it initiates a state transfer for those objects that are out-of-date or corrupt.

The time spent waiting for fetch replies is overlapped with computing digests for each of the partition's objects and comparing those digests with the ones in the partition tree. To obtain the digests for the objects in the current abstract state, the recovering replica must invoke the abstraction function on all the elements of the abstract state vector and compute the respective digests. Those objects whose digests do not match are queued for fetching.

The application may increase the amount of parallelism between state checking and waiting for fetch replies by only partially reconstructing the abstract state in the `restart` upcall, and use the abstraction function to detect if there are data structures that still need to be reconstructed and doing it if necessary. It is more efficient to reconstruct the state in the abstraction function since this is

done in parallel with fetching.

Chapter 4

Example I: File System

This chapter describes the first example application that is used to illustrate our methodology: a replicated file system. The file system is based on the NFS protocol [2]. Its replicas can run different operating systems and file system implementations.

The first three section in this chapter describe the how the successive steps in the methodology were applied in this example: defining the abstract specification; implementing the conformance wrapper; and defining the abstraction function and its inverse. Section 4.4 describes the proactive recovery mechanism in this service and we conclude in Section 4.5 with a discussion of possible optimizations.

4.1 Abstract Specification

The common abstract specification is based on the specification of the NFS protocol [2]. The abstract file service state consists of a fixed-size array of pairs with an object and a generation number. Each object has a unique identifier, *oid*, which is obtained by concatenating its index in the array and its generation number. The generation number is incremented every time the entry is assigned to a new object. There are four types of objects: files, whose data is a byte array; directories, whose data is a sequence of <name, oid> pairs ordered lexicographically; symbolic links, whose data is a small character string; and special *null* objects, which indicate an entry is free. All non-null objects have meta-data, which includes the attributes in the NFS `fattr` structure, and the index (in the array) of its parent directory. Each entry in the array is encoded using XDR [1]. The object with index 0 is a directory object that corresponds to the root of the file system tree that was mounted. Initially, the root directory is empty.

Keeping a pointer to the parent directory is redundant, since we can derive this information by scanning the rest of the abstract state. But it simplifies the inverse abstraction function and the recovery algorithm, as we will explain later.

The operations in the common specification are those defined by the NFS protocol. There are operations to read and write each type of non-null object. The file handles used by the clients are the *oids* of the corresponding objects. To ensure deterministic behavior, we require that *oid*s be assigned deterministically, and that directory entries returned to a client be ordered lexicographically.

The abstraction hides many details; the allocation of file blocks, the representation of large files and directories, and the persistent storage medium and how it is accessed. This is desirable for simplicity and performance. Also, the more we abstract from implementation details such as allocation issues, the grater resilience to software faults due to aging we have (e.g. we tolerate resource leaks since these do not show up in the abstract state and we may fix them during recovery).

4.2 Conformance Wrapper

The conformance wrapper for the file service processes NFS protocol operations and interacts with an off-the-shelf file system implementation using the NFS protocol as illustrated in Figure 4-1. A file system exported by the replicated file service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level relay processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the `invoke` procedure of our replication library, and sends the result back to the NFS client. The replication library invokes the `execute` procedure implemented by the conformance wrapper to run each NFS request.

The conformance rep consists of an array that corresponds to the one in the abstract state but it does not store copies of the objects; instead each array entry contains the type of object, the generation number, and for non-empty entries it also contains the file handle assigned to the object by the underlying NFS server, the value of the timestamps in the object's abstract meta-data, and the index of the parent directory. The rep also contains a map from file handles to *oids* to aid in processing replies efficiently.

The wrapper processes each NFS request received from a client as follows. It translates the file handles in the request, which encode *oid*s, into the corresponding NFS server file handles. Then

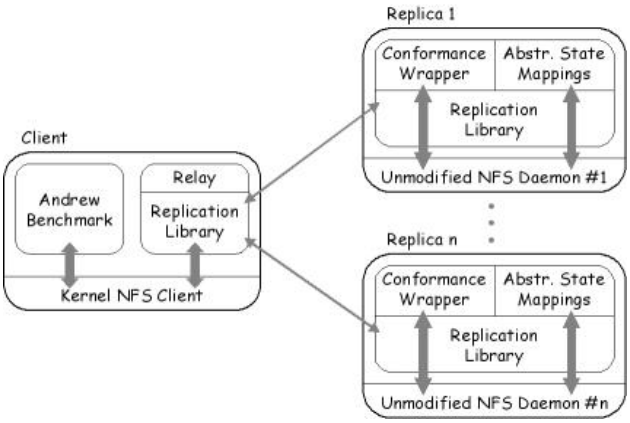

Figure 4-1: Software architecture

it sends the modified request to the underlying NFS server. The server processes the request and returns a reply.

The wrapper parses the reply and updates the conformance rep. If the operation created a new object, the wrapper allocates a new entry in the array in the conformance rep, increments the generation number, and updates the entry to contain the file handle assigned to the object by the NFS server and the index of the parent directory. If any object is deleted, the wrapper marks its entry in the array free. In both cases, the reverse map from file handles to *oids* is updated.

The wrapper must also update the abstract timestamps in the array entries corresponding to objects that were accessed. For this, it uses the value for the current clock chosen by the primary using the `propose_value` upcall in order to prevent the states of the replicas from diverging. However, if a faulty primary chooses an incorrect value the system could have an incorrect behavior. For example, the primary might always propose the same value for the current time; this would cause all replicas to update the modification time to the same value that it previously held and therefore, according to the cache consistency protocol implemented by most NFS clients [10] , cause the clients to erroneously not invalidate their cached data, and thus leading to inconsistent values at the caches of different clients. The solution to this problem is to have each replica validate the choice for the current timestamp using the `check_value` function. In this case, this function must guarantee that the proposed timestamp is not too far from the replica's own clock value, and that the timestamps produced by each primary are monotonically increasing.

Finally, the wrapper returns a modified reply to the client, using the map to translate file handles to *oids* and replacing the concrete timestamp values by the abstract ones.

When handling *readdir* calls the wrapper reads the entire directory and sorts it lexicographically

to ensure the client receives identical replies from all replicas.

4.3 State Conversions

The abstraction function in the file service is implemented as follows. For each file system object, it uses the file handle stored in the conformance rep to invoke the NFS server to obtain the data and meta-data for the object. Then it replaces the concrete timestamp values by the abstract ones, converts the file handles in directory entries to *oids*, and sorts the directories lexicographically.

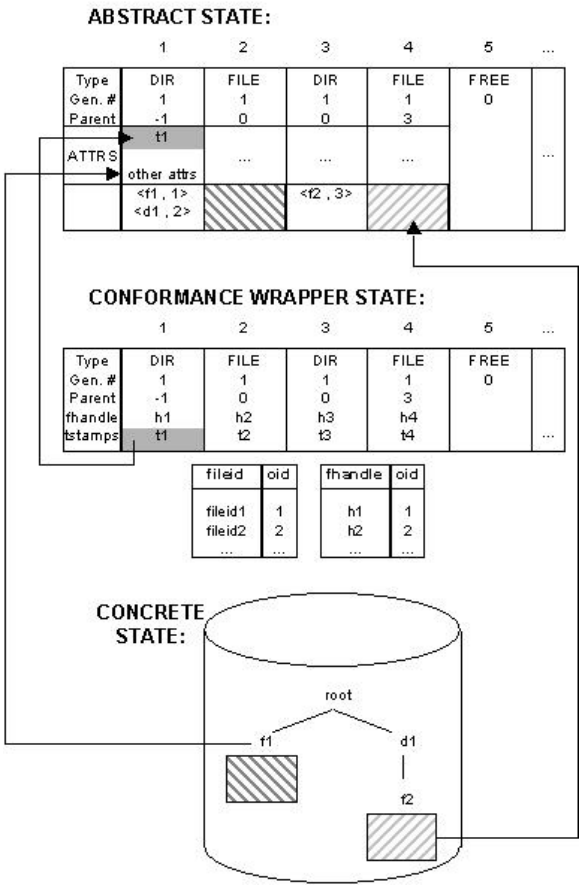

Figure 4-2: Example of the abstraction function

Figure 4-2 shows how the concrete state and the conformance rep are combined to form the abstract state for a particular example. Note that the attributes in the concrete state are combined with the timestamps in the conformance rep to form the attributes in the logical state. Also note that the contents of the files and directories are not stored by the conformance rep, but only in the concrete state.

The pseudocode for the inverse abstraction function in the file service is shown in Figure 4-3.

This function receives an array with the indices of the objects that need to be updated and the new values for those objects. It scans each entry in the array to determine the type of the new object, and acts accordingly.

```
function put_objs(in modified object array)
  for each entry ∈ modified object array do
    if new object's type = file or symbolic link then
      update_directory(new object's parent directory index)
      update object's meta-data in the conformance rep
      set file attributes
      write file's new contents or update link's target
    else if new object's type = directory then
      update_directory(this object's index)
      update object's meta-data in the conformance rep
    else if new object's type = free entry then
      update object's type in the conformance rep

function update_directory(in directory's index)
  if (directory has already been updated or
    directory has not changed) then
    do nothing
  else
    update_directory(new directory's parent directory index)
    read directory's old contents using NFS calls
    for each entry in old directory do
      if entry is not in new directory then
        remove entry using NFS call
      else if entry is in new directory but type is wrong then
        remove entry
    for each entry in new directory do
      if entry is not in old directory then
        create entry using NFS call
```

Figure 4-3: Inverse abstraction function pseudocode

If the new object is a file or a symbolic link, it starts by calling the `update_directory` function, passing the new object's parent directory index as an argument. This will cause the object's parent directory to be reconstructed if needed, and the corresponding object in the underlying file system will be created if it did not exist already. Then it can update the object's entry in the conformance rep, and issue a `setattr` and a `write` to update the file's meta-data and data in the concrete state. For symbolic links, it is sufficient to update their meta-data.

If the new object is a directory it suffices to call `update_directory` passing its own index as an argument, and then updating the appropriate entry in the conformance rep.

Finally, if the new object is a free entry it updates the conformance rep to reflect the new object's

type and generation number. If the entry was not previously free, it must also remove the mapping from the file handle that was stored in that entry to its *oid*. We do not have to update the parent directory of the old object, since it must have changed and will be processed eventually.

The `update_directory` function can be summarized as follows. If the directory that is being updated has already been updated or is not in the array of objects that need to be updated then the function performs no action. Otherwise it calls itself recursively passing the index of the parent directory (taken from the new object) as an argument. Then, it looks up the contents of the directory by issuing a `readdir` call. It scans the entries in the old state to remove the ones that are no longer present in the abstract state (or have a different type in the abstract state) and finally scans the entries in the new abstract state and creates the ones that are not present in the old state. When an entry is created or deleted, the conformance rep is updated to reflect this.

4.4 Proactive Recovery

After a recovery, a replica must be able to restore its abstract state. This could be done by saving the entire abstract state to disk before the recovery, but that would be very expensive. Instead we want to save only the metadata (e.g., the oids and the timestamps). But to do this we need a way of relating the oids to the files in the concrete file system state. This cannot be done using file handles since they can change when the NFS server restarts. However, the NFS specification states that each object is uniquely identified by a pair of meta-data attributes: <fsid,fileid>. We solve the problem by adding another component to the conformance rep: a map from <fsid,fileid> pairs to the corresponding *oids*. The `shutdown` method saves this map (as well as the metadata maintained by the conformance rep for each file) to disk.

After rebooting, the `restart` method performs the following steps. It reads the map from disk; performs a new `mount` RPC call, thus obtaining the file handle for the file system root; and places null file handles in all the other entries in the conformance rep that correspond to all the other objects, indicating that we do not know the new file handles for those objects yet. It then initializes the other entries using the metadata that was stored by the `shutdown` method.

Then the replication library runs the protocol to bring the abstract state of the replica up to date. As part of this process, it updates the digests in its partition tree using information collected from the other replicas and calls `get_obj` on each object to check if it has the correct digest. Corrupt or out-of-date objects are fetched from the other replicas.

The call to `get_obj` will determine the new NFS file handle if necessary. In this case, it goes up the directory tree (using the parent index in the conformance rep) until it finds a directory whose new file handle is already known. Then it issues a `readdir` to learn the names and fileids of the entries in the directory, followed by a `lookup` call for each one of those entries to obtain their NFS file handles; these handles are then stored in the position that is determined by the <fsid,fileid> to *oid* map. Then it proceeds down the path of the object whose file handle is being reconstructed, computing not only the file handles of the directories in that path, but also those of all their siblings in the file system tree.

When walking up the directory tree using the parent indices, we need to detect loops so that the recovery function will not enter an infinite loop due to erroneous information stored by the replica during shutdown.

Computing the new file handles during the object check stage is efficient, since this is done in parallel with fetching the objects that are out-of-date.

Currently, we restart the NFS server in the same file system and update its state with the objects fetched from other replicas. We plan to change the implementation to start an NFS server on a second empty disk and bring it up-to-date incrementally as we obtain the values of the abstract objects. This has the advantage of improving fault-tolerance as discussed in Section 3.2. Additionally, it can improve disk locality by clustering blocks from the same file and files that are in the same directory.

4.5 Discussion

The current design for the abstract state specification has one defect. When a file size grows, the size of the object grows to the same size plus the size of the meta-data. As was mentioned in Section 3.2.2, this can be a problem when those objects become corrupt or out of date at one replica and have to be transferred from some other replica. In particular, if only a small part of the contents of the object is incorrect, we are paying the penalty of transferring a very large object unnecessarily. In Section 3.2.2, we mentioned that a possible solution would be to change the specification for the abstract state such that it uses smaller objects. We exemplify how this could be done for the file system.

An alternative design for the file system abstract state that overcomes this problem is to use an indirection mechanism, where the data stored in objects that correspond to files that are larger than a certain threshold are pointers to data block objects (i.e. their indices in the abstract state vector),

instead of the actual contents of the files.

This would imply storing an additional map in the conformance rep to relate the index of a data block with the corresponding file object. This map would be used in determining which file to read on an invocation of the `get_obj` upcall for a data block object. The inverse map (from file objects to the indices of the data blocks they are using) should also be kept in the conformance rep to allow easier detection of which index of the abstract state array is changed when we write to a file (so it can be passed to the `modify` procedure) and also to speed up the abstraction function for objects that correspond to large files.

This optimization is done at the expense of an increased complexity in the conformance wrapper and state conversions. As we will explain in Chapter 6, this can be problematic since it increases the probability of the conformance wrapper or the state conversions introducing software errors. Since all replicas share this code this induces an undesirable correlation in replica failure.

Chapter 5

Example II: Object-Oriented Database

The methodology described in Chapter 3 was applied to the Thor object-oriented database management system (OODBMS) [36, 35]. In this example, the diversity in the replicas is not obtained from using distinct implementations, but from using a single, non-deterministic implementation.

The first section in this chapter is an overview of the Thor system. Sections 5.2, 5.3, and 5.4 describe the three main steps in the methodology: defining the abstract specification, conformance wrapper and state conversion, respectively. We conclude in Section 5.5 with a discussion of several other aspects of the system that were not included in the current prototype.

5.1 System Overview

Thor is an object-oriented database management system intended to be used in heterogeneous distributed systems to allow programs written in different programming languages to share objects. Thor objects are persistent in spite of failures, and are highly available. Each object has a global unique id. Objects contain data and references to other objects. Thor provides a persistent root for the object universe. An object becomes persistent if it becomes accessible from the persistent root. If it subsequently becomes unreachable from the root, its storage is reclaimed by a distributed garbage collector [42].

Objects are stored at server nodes that are distinct from the machines where application programs reside. The Thor system runs on both the application site and the server site. The component that runs on the server side, called *object repository* (OR), is responsible for managing the storage of persistent objects. For each application there is a dedicated process called *frontend* (FE) that handles all the application requests. The universe of objects is spread across multiple ORs and application

programs can access these objects by interacting with their FEs. The FEs cache and prefetch objects in order to run operations on objects locally, thus reducing the delay as observed by the clients. Thor supports transactions [27] that allow users to group operations so that the database state is consistent in spite of concurrency and failures. Figure 5-1 provides an overview of the system architecture of Thor.

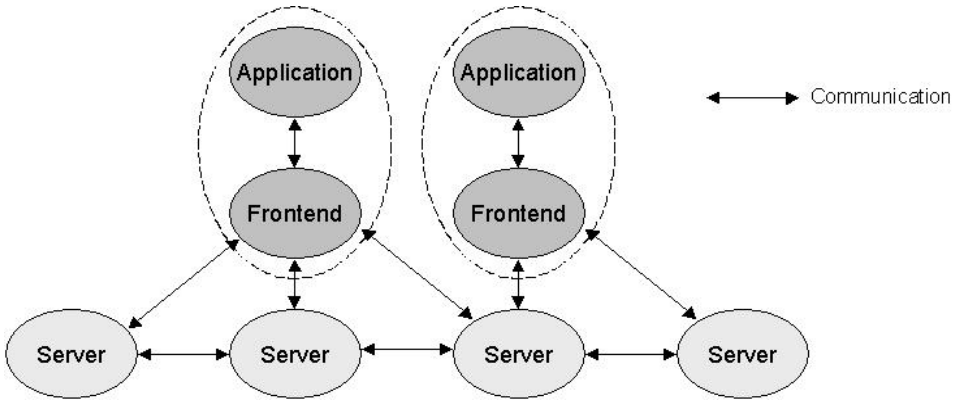

Figure 5-1: Applications, frontends and object repositories in Thor

The system creates a frontend process for an application whenever the latter want to access objects from the Thor universe. When the FE interacts with an OR for the first time, it creates a *session* with that server. The FE and the OR then maintain information about each other until the session terminates. The OR discards all the per-FE information when the respective session ends.

Since the target of our replication technique is going to be the Thor OR, the OR will be the focus the remainder of the system description. A detailed description of the FE can be found in [34, 14].

5.1.1 Object format and fetching

Each OR manages a subset of the Thor object universe. It stores a *root directory* object that contains references to other objects or other directories. Applications can ask for the OR's root directory and access objects from that OR. However, the root directory serves only as an entry point to the universe of persistent objects; objects are subsequently accessed by following pointers between objects or directly using queries.

A fetch command issued to a particular OR identifies the required object by giving its *oref*. An oref is a name local to that particular OR. The *oref* is divided into two parts: a page id and an object number. The page id is used to locate the disk storage for the object's page by looking it up in a table. The first part of the contents of each page includes a table mapping the object number to an

offset within the page. This second level of mapping allows an object to move within the page and change size without affecting its oref. Objects can be identified uniquely by the <OR-id, oref> pair.

Orefs allow objects to refer to other objects at the same OR; objects point to objects at other ORs indirectly via *surrogates*. A surrogate is a small object that contains the identifier of the target object's OR and its oref within that OR.

When an application requests an object that is not in the FE cache, the FE fetches the entire page that contains the object from the corresponding OR. Fetching an entire page improves performance because there is likely to be some locality within a page and therefore other objects on the page are likely to be useful to the FE; also page fetches are cheap to process at both FEs and ORs.

The FE assigns new orefs to newly persistent objects. To avoid conflicts in the values that are assigned by distinct FEs, only one FE has allocation rights for a particular page in each moment. To obtain allocation rights, the FE must invoke an `alloc_page` operation that returns a pageid for a newly created page. The OR keeps track of which FE has allocation rights for each page using a map from page number to the identifier of the FE with allocation rights, or a *null* identifier if there is none.

The FE employs a client caching scheme called Hybrid Adaptive Caching or HAC [14]. HAC is a hybrid between page and object caching that combines the virtues of each — low overheads and low miss rates, respectively — while avoiding their problems. HAC adaptively partitions the FE cache between objects and pages based on the current application behavior: pages in which locality is high remain intact, while for pages in which locality is poor, only hot objects are retained. HAC partitions the FE cache into page frames and fetches entire pages from the OR. To make room for an incoming page, HAC selects some page frames for compaction, discards the cold objects in these frames, and compacts the hot objects to free one of the frames. HAC maintains usage statistics on a per-object basis with low space and time overheads. The technique combines both recency and frequency information to make good replacement decisions.

5.1.2 Transactions

All operations of an application are executed as part of a Thor transaction; a new transaction is started each time the previous one completes. A transaction consists of one or more calls to methods of Thor objects. The application code ends a transaction by requesting a commit or abort. A commit request may fail (causing an abort); if it succeeds, Thor guarantees that the transaction is serialized

with respect to all other transactions and that all its modifications to the persistent universe are recorded reliably. If the transaction aborts, it is guaranteed to have no effect and all its modifications are discarded.

To commit a transaction, a frontend sends the transaction information to one of the ORs and waits for the decision. If all objects used by the transaction reside at that OR, committing can be done locally there; otherwise, this OR, also known as the *coordinator* of the transaction, executes a 2-phase commit protocol [26, 45]. After the outcome of the transaction is known, it informs the FE of the result. The application waits only until phase one of this protocol is over; the second phase is executed in the background.

Aborts can occur for two reasons: stale data in FE caches, and actual conflicts between concurrent transactions. The former is detected using the information in the *frontend invalid set*, whereas the latter is detected using a novel concurrency-control algorithm called *Clock-based Lazy Optimistic Concurrency Control* or CLOCC [6, 5], which will be described below.

The *frontend invalid set* lists pending invalidations for an FE. As soon as an OR knows about a commit of a transaction, it determines what invalidations it needs to send to what FEs. It does this using a *cached pages directory*, which maps each page in the OR to the set of FEs that have cached copies of that page. When a transaction commits, the OR adds a modified object to the invalid set for each FE that has been sent that object's page. An object is removed from an FE's invalid set when the OR receives an ack for the invalidation from the FE. The hybrid caching scheme described above allows the FE to discard the invalid object while retaining the remaining objects of the page. A page is removed from the cached pages directory for the FE when the OR is informed by the FE that it has discarded the page.

All messages of the invalidation protocol are piggybacked in the other messages, which reduces the communication costs associated with this protocol. To avoid long delays in sending invalidation protocol messages when there are no requests, periodic "I'm alive" requests are sent from the FEs and replied immediately.

Concurrency control is done at the level of objects to avoid false conflicts [11]. Furthermore, Thor uses an optimistic concurrency-control algorithm to maximize the benefit of the client cache and avoid communication between FEs and ORs for concurrency control purposes. The FE runs a transaction assuming that reads and writes of the objects in cache are permitted. When the transaction attempts to commit, the FE informs an OR about the reads and writes done by the transaction, together with the new values of any modified objects. The FE assigns the transaction a timestamp,

that is obtained by reading the time of its local clock and concatenating it with the *or-id* of the OR that is being contacted to obtain a unique number. It is assumed that the clock values of all nodes in the system are loosely synchronized to within a few tens of milliseconds of one another. This assumption is not needed for correctness, but improves performance since it allows each OR to make time-dependent decisions, e.g. when discarding old information. This assumption is a reasonable one for current systems [44].

Every participant of the transaction will try to serialize the transaction relative to other transactions. This is done using a backward validation scheme [28]: the committing transaction is compared with other committed and committing transactions, but not with active transactions (since that would require additional communication between FEs and ORs).

A transaction's timestamp determines its position in the serial order, and therefore the system must check whether it can commit the transaction in that position. For it to commit in that position the following conditions must be true:

1. For each object it used (read or modified), it must have used the latest version, i.e., the modification installed by the latest committed transaction that modified that object and that is before it in the serialization order.

2. It must not have modified any object used by a committing or committed transaction that follows it in the serialization order. This is necessary since otherwise we cannot guarantee condition (1) for that later transaction.

An OR validates a transaction using a *validation queue,* or VQ, and the object sets from the *frontend invalid set*. The VQ stores the read object set (ROS) and modified object set (MOS) for committed and committing transactions. For now we assume that the VQ can grow without bounds; we discuss how entries are removed below. If a transaction passes validation, it is entered in the VQ as a committing transaction; if it aborts later it is removed from the VQ while if it commits, its entry in the VQ is marked as committed.

A participant validates the transaction T by (1) checking whether any objects used by a transaction T are in the invalid set for T's client; and (2) comparing T's MOS and ROS against those of all the transactions in the validation queue to see if the conditions above are violated. A complete description of this check can be found in [6, 5].

Time is also used to keep the VQ small. A *threshold timestamp* is maintained, VQ.t, and the VQ does not contain any entries for committed transactions whose timestamp is less than VQ.t. An

attempt to validate a transaction with timestamp smaller than VQ.t will fail, but this is an unlikely situation given the loosely synchronized clocks assumption. VQ.t is set to the current time minus some δ that is large enough to make it highly likely that prepare messages for transactions for which this OR is a participant will arrive when their timestamp is greater than the threshold.

When the transactions prepare and commit the usual information (the ROS, MOS, and new versions of modified objects) is written to the transaction log. This is necessary to ensure that effects of committed transactions survive failures.

5.1.3 The Modified Object Buffer

The fact that invalid objects are discarded from the client cache while other objects in that page are retained means that it is impossible for FEs to send complete pages to the OR when a transaction commits. Instead object shipping must be employed. However, ultimately the OR must write objects back to their containing pages in order to preserve clustering. Before writing the objects it is usually necessary to read the containing page from disk (this is called an *installation read*). Doing an installation read while a transaction commits degrades performance, so another approach is needed.

The approach that is employed uses a volatile buffer in which recently modified objects are stored; this buffer is called *modified object buffer*, or MOB [24]. When modified objects arrive at the OR, they are stored in the MOB instead of being installed in a page cache. The modifications are written back to disk lazily as the MOB fills up and space is required for new modifications. Only at this point are installation reads needed.

The MOB architecture has several advantages over a conventional page buffer. First, it can be used in conjunction with object shipping, and yet installation reads can be avoided when transactions commit. Second, less storage is needed to record modifications in the MOB than to record entire modified pages in a page cache. Therefore the MOB can record the effects of more transactions than a page cache, given the same amount of memory; as a result, information about modifications can stay in the MOB longer than in a page cache. This means that by the time we finally move an object from the MOB to disk, there is a high probability that other modifications have accumulated for its page than in the case of a page cache. We call this effect write absorption. Write absorption leads to fewer disk accesses, which can improve system performance.

Now we describe how the MOB works; more information can be found in [24]. The OR contains some volatile memory, disk storage, and a stable transaction log as shown in Figure 5-2. The disk provides persistent storage for objects; the log records modifications of recently committed

transactions. The volatile memory is partitioned into a page cache and a MOB. The page cache holds pages that have been recently fetched by FEs; the MOB holds recently modified objects that have not yet been written back into their pages on disk.

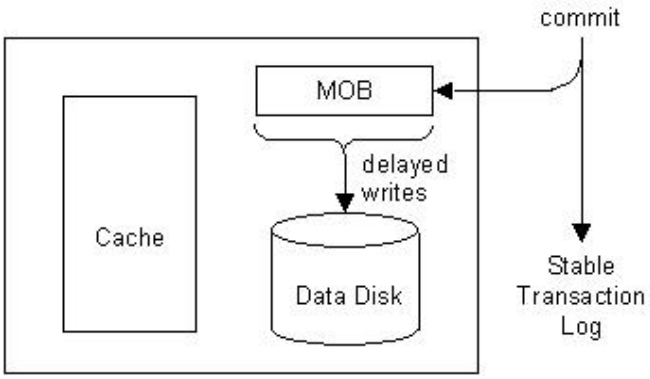

Figure 5-2: Server organization

Modifications of committed transactions are inserted into the MOB as soon as the commit has been recorded in the log. They are not written to disk immediately. Instead a background flusher thread lazily moves modified objects from the MOB to disk. The flusher runs in the background and does not delay commits unless the MOB fills up completely with pending modifications.

Pages in the cache are completely up-to-date: they contain the most current versions of their objects. However, pages on disk do not reflect modifications of recent transactions, i.e., modifications in the MOB. Therefore a fetch request is processed as follows: If the needed page is not in the page cache, it is read into the cache and then updated to reflect all the recent modifications of its objects in the MOB. Then the page is sent to the FE. When the page is evicted from the cache, it is not written back to disk even if it contains recent modifications; instead we rely on the flusher to move these modifications to disk.

The page cache is managed using a LRU policy, but the MOB is managed in a FIFO order to allow the log to be truncated in a straightforward way. The flusher scans the MOB and identifies a set of pages that should be written to disk to allow a prefix of the log to be truncated. This set of pages is read into the page cache if necessary (these are the installation reads). Then all modifications for those pages are installed into the cached copies and removed from the MOB. Then the pages are written to disk, and finally a prefix of the log is truncated. Therefore the operation of the flusher replaces the checkpointing process used for log truncation in most database systems [27]. The log may contain records for modifications that have already been installed on disk but this is not a problem: the modifications will be re-installed if there is a failure (and re-entered in the MOB),

but failures are rare so that the extra cost to redo the installs is not an issue and correctness is not compromised since installation is idempotent.

5.2 Abstract Specification

Now we will focus on how we applied the methodology to build a replicated version of the Thor database. We will begin by defining the abstract specification for the database, then we describe the conformance wrapper and the abstraction functions for this system. This description is based on a single OR version of the system. The extensions required to support multiple ORs are discussed in Section 5.5.

The abstract specification for the object-oriented database is based on the state and operations of Thor, since in this case the replicated system is not meant to use different implementations, and there is no standard way to access the state of an object-oriented database.

The operations defined in the abstract specification are those defined in the FE/OR interface of Thor, and the semantics of the operations is the same as defined by Thor. These are the following.

- `fetch` — This operation requests the page that contains the object whose oref is passed as an argument. It is assumed that the FE will cache the page that is being fetched.
- `alloc_page` — This is used to allow the FE to request allocation rights for a new page. Returns the page id for the new page or an error code.
- `get_root` — Has no arguments and returns the oref of the root object.
- `commit` — Tries to commit a transaction, passing as arguments the NOS, ROS and MOS (new, read and written object sets, respectively) and the timestamp for this transaction. Returns the result of the transaction (commit or abort) and possibly an error code indicating why it aborted.
- `"I'm Alive"` — Exchanged when there is no activity for a long period of time.
- `Start / End Session` — These messages are sent by the FE to initiate and end sessions with a particular OR. This causes the OR to create or free, respectively, the data structures that store all information that it needs to manage a session for that particular FE. This information includes the frontend invalid set and all the information about that particular FE in the cached pages directory.

All requests and replies may include additional information related to the invalidation protocol.

The only operation that is omitted from the abstract specification is the `or_stats` operation, which has no input arguments and retrieves a set of statistics from the OR, such as the amount of time spent in each operation. It was omitted since different replicas have different values for these statistics, and thus a common abstract operation is not the correct way to retrieve these values. This operation is meant to be used for debugging and profiling, and is really not part of the system.

The common abstract state is defined as follows. The array of objects is divided in four fixed-size subarrays. These define the following parts of the abstract state.

- **OR pages** — This defines the contents of the database. The OR objects are already clustered in pages, so the contents of the entries in the abstract state are defined to be exactly the same as the pages that are stored in disk. Also, OR pages are assigned page numbers deterministically; this number will correspond to the position in the subarray of the abstract state that describes the OR pages. Pages that have not been allocated can be identified by placing the invalid OR number 0 in the page header. The first few bytes of each page contain a table that maps from object numbers to their offsets within the page; this table is also included in the abstract object.

- **Validation Queue** — The validation queue must be included in the abstract state because it defines the outcome of future transactions that use objects that were also used by committing or recently committed transactions. To efficiently represent a queue as an array of objects, those objects should not be kept in order at the begining of the array. This is so because an unnecessarily large number of objects would be changed when the queue is truncated (namely those objects that remained in the queue but had their position in the array changed); this works inefficiently with the incremental checkpointing scheme employed by the library, since only the objects that changed since the last checkpoint have to be copied to a checkpoint. Instead, we define that the position of a validation queue entry in the abstract state subarray does not change while that entry remains in the validation queue. A deterministic procedure is defined to assign a free position in the subarray to a new entry in the validation queue. The entries in the abstract state that correspond to VQ entries contain the timestamp that was assigned to that entry, the status of that transaction, the number of objects that were read and written by that transaction, followed by the corresponding *oref*s. Free entries in this abstract state sub-array contain a special *null* timestamp. The first entry in this subarray is a special

entry that contains the validation queue threshold.

- **Frontend invalid set** — This structure lists pending invalidations for each FE, and is used in determining if a transaction has to abort due to having read stale versions of objects that were changed by concurrent transactions. Therefore, this information is also included in the abstract state. The abstract state for the frontend invalid set consists of an array with length equal to the maximum number of FEs in the system. Each entry in the array contains the FE identifier (or an invalid null identifier for free entries), the number of invalid objects followed by the corresponding orefs. When an FE starts a session with a particular OR, it is assigned a client number that is used to determine the position of its invalid set in this sub-array of the abstract state.

- **Cached pages directory** — This keeps track of the set of FEs that have cached copies of each page in the OR. It is used in determining the new frontend invalid sets after a transaction commit. There is one entry in the array per OR page. As in the case of the OR page subarray, the position of an entry in this subarray is equal to the page number. Each entry contains a count of the FEs that have cached the page, followed by a list with the FE identifier and the state (up-to-date or not) of the page they are holding. We also record here the identifier of the FE that has allocation rights for this page.

The abstraction hides the details of caching and lazy modification write back at the server side, giving a single view of the database contents. It therefore allows different replicas to cache different pages, or write back objects to pages at different times, without having their abstract states diverge.

5.3 Conformance Wrapper

Unlike in the file system example, the conformance wrapper for the Thor object-oriented database is not a separate process that invokes operations on an unmodified server. Instead we interposed the wrapper code directly on that implementation.

As explained in Section 5.1, the FE assigns each transaction a timestamp that is used for concurrency control purposes. A faulty FE could cause an excessive number of aborts in subsequent transactions by assigning a timestamp that is too large. This is so because all future transactions will be validated against this one and, assuming they have smaller timestamp values, they will abort if they read or modified any object that was written by this transaction. To avoid this, when validat-

ing a transaction, the OR must abort it if the timestamp that is proposed is larger than the server's current time plus a tolerance constant τ. Since the replicas will not agree on the local clock reading used to perform the check, we must use the `propose_value` upcall to obtain a unique local clock value for all the replicas.

As in the previous example, a faulty primary might subvert the system's desired behavior by proposing an erroneous value for the current time, which might allow a transaction with an excessive timestamp to commit. To avoid this situation, all replicas must use the `check_value` upcall to verify that the value that is proposed by the primary is within a range of δ of their current clock reading. As usual, if enough replicas refuse an erroneous value that is proposed by a faulty primary, the request will not be able to execute, and eventually a timeout mechanism will cause the replicas to start the view change procedure and a new primary will be chosen.

There is another check that the conformance wrapper needs to perform due to the fact that FEs are no longer trusted. A malicious FE could exploit the fact that FEs decide the values for the orefs of newly created objects to lead the database to an inconsistent state by either proposing an oref value of an already existing object or by trying to create an object that does not fit in the page that it is being placed. Therefore, the OR must verify if any of these conditions is violated, and if so abort the correspondent transaction.

It does this by keeping a small additional amount of information on the cached pages directory. In this structure, the OR already keeps track of which FE has allocation rights for this page. It must also maintain information about the free space available in the page and the next object number to be assigned (the system must now impose that object numbers are assigned sequentially). When an FE creates an object, the OR checks if it has allocation rights for the corresponding page, if the object fits the page and if it has been assigned a valid object number. If any of these conditions is violated, the transaction that created the object is aborted.

In order to prevent a malicious client from requesting allocation rights for the entire database, we must also limit the amount of pages that each FE can be granted these rights.

The conformance rep maintains two data structures: an array of size equal to the size of the validation queue subarray in the abstract state, and another of size equal to the maximum number of FEs in the system.

The entries in the first array are the timestamps of the VQ entries that correspond to the entry with the same index in the abstract state subarray. The entries in the second array are pointers to the data structure that Thor uses to maintain all the per-FE information. These two arrays are used in

the state conversions as described in Section 5.4.

The inverse correspondence (from objects in the concrete state to indices in the abstract state array) is also maintained in the conformance rep to speed up the process of knowing which index to pass as an argument to the `modify` procedure of the BASE library whenever that part of the state is modified.

5.4 State Conversions

The abstraction function maps from the concrete state to the abstract state and is implemented in the `get_obj` upcall used by the BASE library. This upcall receives as an input parameter the index in the abstract state array of the object that the library is trying to retrieve. So this function must begin by determining which subarray the requested object belongs to. Then, the object in the abstract state is computed accordingly:

- **OR page** — To compute this kind of object, we must first install all pending modifications in that page. To do so, we invoke an existing OR method that retrieves all modifications for a particular page, installs them and returns the updated version of that page. By doing this we are not flushing the MOB, but only temporarily installing the modifications that belong to the page that is being requested.

- **Validation queue** — If the object is in the validation queue subarray, and it is the first entry in the sub-array, that means the object requested only holds the validation queue threshold, so we just read that value and copy it to the object that is being returned. If it is a VQ entry, we first retrieve the timestamp that corresponds to that entry from the timestamps array in the conformance rep, and then fetch that entry from the VQ. Then, we just copy the number of objects read and modified by the transaction and their *oref*s to compose the object in the abstract state.

- **Frontend invalid set** — We begin by retrieving the data structure in the Thor implementation that holds all the per-FE information using the respective array in the conformance rep. Then we copy the FE identifier and the number of invalid objects with the corresponding orefs to the newly created abstract object.

- **Cached pages directory** — In this case, we determine the page number that the requested directory entry corresponds to by computing the offset to the beginning of the subarray, and

then we look up all the information about which FEs have cached this page, and the current status of the cached page from the table. We also copy the identifier of the FE with allocation rights to the page, that is stored in the same directory.

The inverse abstraction function works as follows. It receives an array with new values for the abstract objects. We must iterate through the elements of that array and restore their state. Again, there are four possible cases according to the type of object that we are restoring.

- **OR page** — If the contents of the new page object indicate that it no longer exists, we remove the page from the OR, freeing up any resources that it had allocated. Otherwise, before we update an entry that corresponds to an OR page that already existed we must remove the entries in the MOB that correspond to that page, so that it has no pending modifications and therefore the contents that we are going to install in the OR pages are not going to be clobbered by those modifications. Then we fetch the page, placing it in the OR cache, and then overwrite the contents with the ones we receive from the function, and force the page to be written back to disk.

- **Validation queue, frontend invalid set and cached pages directory** — If the entry already exists and it remains present in the new abstract state, we just update its value according to the new abstract object value. Otherwise we must delete the entry if the new abstract object describes a non-existent entry, or create the entry if it did not previously exist and fill in the values that are described by the new abstract object. The conformance rep must be updated accordingly.

5.5 Discussion

In this section we discuss several features of our system that are not included in the current prototype but are expected to be implemented in the future. We present here their possible design.

5.5.1 Multiple Server Architecture

The current prototype only allows one OR in the Thor universe. We discuss here how the system described above could be extended to support multiple ORs.

The only limitation of the current system that prevents us from having multiple ORs is that it does not provide support for distributed transactions, i.e. transactions that use objects from more

than one OR. As we mentioned in Section 5.1, in a distributed transaction the client will send a commit request to one of the ORs that is involved in the transaction. This OR, also known as the *coordinator* of the transaction, executes along with the remaining ORs (called the *participants*) a 2-phase commit protocol and informs the FE of the result.

Therefore, we must extend the interface in Section 5.2 to support the 2-phase commit protocol operations: `prepare` and `commit`. These operations are not invoked by an FE, but by a replicated OR, i.e. there is a new scenario where a group of replicas (the coordinator OR) act as a client of another replicated OR (one of the participants).

If each of the replicas in the coordinator OR sent independently a request to the participant OR the number of messages with be prohibitively large. In fact, a simple analysis can prove that this number is $O(n^2)$, where n is the number of replicas in each OR.

A scalable solution to this problem is proposed in [7]. It is based on having the primary of the client group perform the request on behalf of the other replicas in its group. After executing the request, it sends the reply to all the backups. As usual, we must prevent a faulty primary from causing the system to misbehave, namely, we have to prevent a faulty primary from (1) forging a request that will cause the participant OR to change to an incorrect state (e.g. requesting a two phase commit that was not invoked by a client); (2) forging the reply of the participant OR that is communicated to the remaining replicas in the coordinator OR; and (3) not executing the operation, causing the system to stall.

The first problem is solved imposing that all read-write requests from replicated clients have to be sent along with a set of $f + 1$ message authentication codes [9] (or MACs) for that request from other replicas, that provide proof that at least one non-faulty replica is invocating the request.

The second condition can also be assured using MACs. In the ordinary case (when there is a single client), after executing a request, each replica authenticates the reply using the key of a single client. When the client is a primary acting on behalf of a group of replicas, we impose that the replicas that execute the request must authenticate the reply using the keys for each of the replicas in the client group, which allows the backup replicas in that group to verify the authenticity of the reply.

Finally, the third condition is taken care of by the view change mechanism that will cause the faulty primary to be replaced if the request does not get executed after a certain time limit.

With this scheme, the number of messages that is exchanged per request in normal case operation (i.e. when the primary is non faulty) is only $O(n)$.

5.5.2 Proactive Recovery

After recovery, we must be able to restore the abstract state of the database. The conformance wrapper must save enough state to disk during shutdown to allow this to happen. We discuss how this could be done separately for each subset of the abstract state.

- **OR pages** — For this part of the state there are two distinct possibilities for recovery. In the first scheme, we drain the MOB during shutdown, causing all pending modifications to be written to disk. In this case, during restart the OR pages are read from disk and we compute digests over the values that we read. The other possibility is to save the MOB to disk during shutdown. During recovery, this is read from disk together with the OR pages, and the pending modifications are installed then. The second scheme has the advantage of avoiding installation of pending modifications during shutdown, which can be more costly than saving the MOB to disk. But at restart it has the penalty of additional reads from disk, and installation of modifications in the MOB. This penalty could although be minimized if these operations are done in parallel with fetches.

- **Validation Queue** — Since this part of the state changes very rapidly, it may not be worth saving to disk during shutdown. In fact, if the time to reboot is larger than the validation queue threshold plus the tolerance in the future τ, we can guarantee that all existing VQ entries will have been removed from the VQ when the replica restarts, and therefore saving them to disk is pointless. Thus, the best solution is to save no information to disk concerning the VQ, and declare the VQ to be empty after restart, so that only to non-null positions will have to be fetched.

- **Frontend invalid set and Cached pages directory** — These represent a small amount of information, and therefore we should be able to save all these structures to disk with low cost.

5.5.3 Garbage Collector

Thor includes a garbage collection mechanism to reclaim storage used by objects that are no longer reachable from the persistent root [40, 42, 41, 43]. This scheme partitions the memory into independently collectible areas to allow better scalability. Any policy can be used to select which partition to collect next.

Garbage collection updates the contents of OR pages by removing objects that are no longer

reachable. Consequently, it changes the abstract state, as it was defined it in Section 5.2. Since all replicas must agree on the contents of their abstract state after executing the same sequence of operations, GC cannot be done independently at each replica, nor in parallel with the execution of FE requests (e.g., using a background thread that collects unreachable objects).

We cannot change the abstract state definition to only include reachable objects either, since that would imply performing GC on every operation, which would be too expensive.

The solution we propose is to define garbage collection as an operation that is invoked by the primary and executed by all replicas using the normal protocol. This operation takes as an argument the identifier of the partition to garbage collect. This way, we can serialize GC with the remaining operations and keep all replicas agreeing on the abstract state.

As usual, we must prevent a faulty primary from doing wrong things like starting the garbage collector more often than necessary or not invoking the garbage collection operation for a long time. To detect this, replicas need to agree on a deterministic heuristic that determines when and which partition to garbage collect.

We intend to use the validation of non-deterministic choices mechanism described in Section 3.2 to assure correct garbage collection operation. When a garbage collection operation is proposed, all replicas use the `check_value` upcall to verify if the garbage collection operation is being invoked at the correct time, i.e., the heuristic that triggers garbage collection confirms the need to perform it at that moment, and that the correct partition is going be updated. Also, when the primary does not invoke the garbage collection operation for an excessive time, the other replicas may use this upcall to prevent any other operation from being executed.

This way, when a faulty primary tries to cause incorrect or slower functioning of the system by invoking too many or too few garbage collection operations, the non-faulty replicas will detect it and reject the `check_value` call. Consequently, the request will not be executed and the view change mechanism will cause a new primary to be selected.

This GC algorithm uses an additional set of data structures that we did not define before [42]. Each partition must keep an inlist and an outlist. These are the objects that are referenced from other partitions and objects that reference objects in other partitions. This information can be included in the abstract state, but is redundant. It can be derived from the global view of the database. Still, including it in the abstract state can speed up the inverse abstraction function, since that information does not have to be derived.

Chapter 6

Evaluation

Our replication technique must achieve two goals to be successful: it must have low overhead when compared to the implementations that it uses; and the code of the conformance wrapper and the state conversion functions must be simple. It is important for the code to be simple to reduce the likelihood of introducing more errors and to keep the monetary cost of using our technique low.

This chapter presents results to evaluate the performance and code complexity of the systems that were described in Chapters 4 and 5. The results show that our replicated file system performs comparably with the off-the-shelf, unreplicated NFS implementations it wraps; that the Byzantine fault-tolerant version of the Thor object-oriented database performs comparably to its previous implementation; and that in both applications the code added by the BASE technique is simple when compared to the code of the original implementations.

This Chapter is divided in two parts. Section 6.1 shows the evaluation results for the replicated file system and Section 6.2 describes the evaluation of the replicated object-oriented database.

6.1 Replicated File System

This section presents results of our experiments for the replicated file system. In Section 6.1.1 we compare the performance of our replicated file system against the off-the-shelf, unreplicated NFS implementations. In Section 6.1.2 we show that the wrapper and mappings that were introduced by the BASE technique are simple when compared to the original implementations.

6.1.1 Overhead

Experimental Setup

Our technique has three advantages: reuse of existing code, software rejuvenation using proactive recovery, and opportunistic N-version programming. We ran experiments with and without proactive recovery in a *homogeneous* setup where all replicas ran the same operating system; these experiments measured the overhead of our technique in systems that benefit from the first two advantages. We also ran experiments without proactive recovery in an *heterogenous* setup where each replica ran a different operating system to measure the overhead in systems that benefit from the first and third advantages.

All experiments ran with four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications.

In the homogeneous setup, clients and replicas ran on Dell Precision 410 workstations with Linux 2.2.16-3 (uniprocessor). These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. The machines were connected by a 100 Mb/s switched Ethernet and had 3Com 3C905B interface cards. Each machine was connected by a single Category 5 cable to a full-duplex port in an Extreme Networks Summit48 V4.1 switch. This is a store-and-forward switch that can forward IP unicast and multicast traffic at link speed. Additionally, it performs IGMP snooping such that multicast traffic is forwarded only to the members of the destination group. The experiments ran on an isolated network, and we used the Pentium cycle counter to measure time accurately.

In the heterogeneous setup, the client and one of the replicas ran on machines identical to the ones in the homogeneous setup. The other replicas ran on different machines with different operating systems: one ran Solaris 8 1/01; another ran OpenBSD 2.8; and the last one ran FreeBSD 4.0. All these machines had Pentium III processors with frequencies between 600 and 700 MHz, had between 256 and 512 MB of memory; and had disks with performance comparable to the model mentioned above. The machines were connected by a 100 MB/s network but they were connected to different switches. The network was not isolated and had some load. In these experiments, we had to disable the IP multicast capabilities of the library since we were not able to perform IP multicast across the different subnetworks that these machines were connected to.

The BASE library was configured as follows. (For a complete explanation of these parameters see [13].) The checkpoint period K was 128 sequence numbers, which causes garbage collection of

the log to occur several times in each experiment. The size of the log L was 128 sequence numbers. The state partition tree had 4 levels and each internal node had 256 children. Requests for operations with argument sizes greater than 255 bytes were transmitted separately; the others were inlined in pre-prepares. The digest replies optimization was not applied when the size of the operation was less than or equal to 32 bytes. The window size for request batching was set to 1.

All experiments ran the modified Andrew benchmark [30, 46], which emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files. Unfortunately, the Andrew benchmark is too small for current systems and therefore it does not exercise the NFS service. So we increased the size of the benchmark by a factor of n similiarly to what was done in [17]. In this scaled up version of the benchmark, phase 1 and 2 create n copies of the source tree, and the other phases operate in all these copies. We ran a version of Andrew with n equal to 100, Andrew100, that creates approximately 200 MB of data and another with n equal to 500, Andrew500, that creates approximately 1 GB of data. Andrew100 fits in memory at both the client and the replicas but Andrew500 does not.

The benchmark ran at the client machine using the standard NFS client implementation in the Linux kernel with the following `mount` options: UDP transport, 4096-byte read and write buffers, allowing write-back client caching, and allowing attribute caching. All the experiments report the average of three runs of the benchmark and the standard deviation was always below 7% of the reported values.

Homogeneous

Tables 6.1 and 6.2 present the results for Andrew100 and Andrew500 in the homogeneous setup with no proactive recovery. They compare the performance of our replicated file system, BASEFS, with the standard, unreplicated NFS implementation in Linux with Ext2fs at the server, NFS-std. In these experiments, BASEFS is also implemented on top of a Linux NFS server with Ext2fs at each replica.

The results show that the overhead introduced by our replication technique is low: BASEFS takes only 26% longer than NFS-std to run Andrew100 and 28% longer to run Andrew500. The overhead is different for the different phases due to variations in the amount of time the client spends computing between issuing NFS requests.

phase	BASEFS	NFS-std
1	0.9	0.5
2	49.2	27.4
3	45.4	39.2
4	44.7	36.5
5	287.3	234.7
total	427.65	338.3

Table 6.1: Andrew100: elapsed time in seconds

There are three main sources of overhead: (1) converting arguments and results and updating the conformance rep; (2) maintaining checkpoints of the abstract file system state; and (3) the cost of running the Byzantine-fault-tolerant replication protocol. The Byzantine fault-tolerance protocol represents 77% of the total overhead, checkpointing is 17%, and argument and result conversion is only 6%. Most of the time (74%) spent in checkpointing is invoking the abstraction function (`get_obj`). These results show that the overhead introduced by the conformance wrapper and state conversions are quite low relative to the overhead of the algorithm.

phase	BASEFS	NFS-std
1	5.0	2.4
2	248.2	137.6
3	231.5	199.2
4	298.5	238.1
5	1545.5	1247.1
total	2328.7	1824.4

Table 6.2: Andrew500: elapsed time in seconds

We also ran Andrew100 and Andrew500 with proactive recovery. The results, which are labeled BASEFS-PR, are shown in Table 6.3. The results for Andrew100 were obtained by recovering replicas round robin with a new recovery starting every 80 seconds, and reboots were simulated by sleeping 30 seconds. We obtained the results for Andrew500 in the same way but in this case a new recovery was started every 250 seconds and reboots were simulated by sleeping 60 seconds. This leads to a window of vulnerability of approximately 6 minutes for Andrew100 and 17 minutes for Andrew500; that is, the system will work correctly as long as fewer than 1/3 of the replicas fail in a correlated way within any time window of size 6 (or 17) minutes. The results show that even with these very strong guarantees BASEFS is only 32% slower than NFS-std in Andrew100 and 30% slower in Andrew500.

system	Andrew100	Andrew500
BASEFS-PR	448.2	2375.8
BASEFS	427.65	2328.7
NFS-std	338.33	1824.4

Table 6.3: Andrew with proactive recovery: elapsed time in seconds.

Table 6.4 presents a breakdown of the time to complete the slowest recovery in Andrew100 and Andrew500. *Shutdown* accounts for the time to write the state of the replication library and the conformance rep to disk, and *restart* is the time to read this information back. *Check* is the time to rebuild the *oid* to file handle mappings in the conformance wrapper, convert the state stored by the NFS server to its abstract form and verify that the digests of the objects in the abstract state match the digests in the partition tree that were stored before shutdown. *Fetch* is the time to fetch out-of-date objects. This is done in parallel with converting and checking the state, so part of the fetch protocol is taking place when the *check* time is being measured.

	Andrew100	Andrew500
shutdown	0.07	0.51
reboot	30.05	60.09
restart	0.19	1.89
check	11.00	144.57
fetch	7.89	30.43
total	49.20	237.49

Table 6.4: Andrew: maximum time to complete a recovery in seconds.

The recovery time in Andrew100 is dominated by the time to reboot but as the state size increases, reading, converting, and checking the state becomes the dominant cost; this accounts for 145 seconds in Andrew500 (61% of the total recovery time).

In the worst-case Andrew500 recovery shown above, the amount of data fetched during recovery (i.e. the total size of the objects that were out-of-date) was over 100 MBytes. These results encourage us to think that the performance degradation in the presence of software faults will be limited. This is so because most of the time in recovery is spent checking the current state, which will probably not increase in the presence of software faults. Furthermore, the amount of data transferred in a recovery for the Andrew500 benchmark is so large that, even if the error caused the entire state to be corrupt, the penalty to transfer it would not be much larger than what is shown in the results.

As mentioned, we would like our implementation of proactive recovery to start an NFS server on a second empty disk with a clean file system to improve the range of faults that can be tolerated. Extending our implementation in this way should not affect the performance of the recovery significantly. We would write each abstract object to the new file system asynchronously right after checking it. Since the value of the abstract object is already in memory at this point and it is written to a different disk, the additional overhead should be minimal.

Heterogeneous

Table 6.5 presents results for Andrew100 without proactive recovery in the heterogenous setup. In this experiment, each replica in BASEFS runs a different operating system with a different NFS and file system implementation. The table also presents results for the standard NFS implementation in each operating system.

system	elapsed time
BASEFS	1753.2
FreeBSD	1162.9
OpenBSD	1097.8
Solaris	1007.4
Linux	338.3

Table 6.5: Andrew100 heterogeneous: elapsed time in seconds

Since operations cannot complete without the cooperation of at least 3 replicas, the overhead must be computed relative to OpenBSD which has the third fastest time. The overhead of BASEFS in this experiment is 60%. The higher overhead is not intrinsic to our technique but is an artifact of our experimental setup: we were unable to use IP multicast when running the Byzantine fault tolerance library in this setup, and the machines were connected to two switches with some external load on the link connecting them.

The substantial difference between the performance of the Linux implementation of NFS and the performance in the remaining operating systems is explained by the fact that the NFS implementation in Linux does not ensure stability of modified data and meta-data before replying to the client (as required by the NFS protocol [53]). An interesting fact about our performance results is that if we compare the performance of our homogeneous setup for BASEFS shown in Table 6.1 (which ensures stability of data before replying by replicating it) with the non-replicated implementations that also ensure stability of modified data (the FreeBSD, OpenBSD and Solaris implementations),

we conclude that those implementations are $136 - 172\%$ slower than BASEFS-Linux.

6.1.2 Code Complexity

To implement the conformance wrapper and the state conversion functions, it is necessary to write new code. It is important for this code to be simple so that it is easy to write and not likely to introduce new bugs. We measured the number of semicolons in the code to evaluate its complexity. Counting semicolons is better than counting lines because it does not count comment and blank lines.

The code has a total of 1105 semicolons with 624 in the conformance wrapper and 481 in the state conversion functions. Of the semicolons in the state conversion functions, only 45 are specific to proactive recovery. To put these numbers in perspective, the number of semicolons in the code in the Linux 2.2.16 kernel that is directly related with the file system, NFS, and the driver of our SCSI adapter is 17735. Furthermore, this represents only a small fraction of the total size of the operating system.

6.2 Replicated Object-Oriented Database

This section presents results of our experiments for the replicated version of the Thor object-oriented database. Again, we divide this in two parts. Section 6.2.1 shows the overhead introduced by replication and Section 6.2.2 shows the complexity of the wrapper and mappings that were introduced by the BASE technique.

6.2.1 Overhead

Experimental Setup

The experimental setup was similar to the homogeneous setup that was described in Section 6.1.1 for the BASEFS system, and the BASE library was also configured in a similar way. We compared the our version developed using the BASE technique against the original version of the Thor database, which also uses replication to achieve persistence. It uses a much simpler primary-backup replication scheme [37]. It was configured with one primary and two backups, and consequently it tolerates a single benign fault (i.e., a faulty replica may not produce incorrect results).

Our workloads are based on the OO7 benchmark [12]; this benchmark is intended to match the characteristics of many different CAD/CAM/CASE applications. The OO7 database contains a tree

of assembly objects, with leaves pointing to three composite parts chosen randomly from among 500 such objects. Each composite part contains a graph of atomic parts linked by connection objects; each atomic part has 3 outgoing connections. All our experiments ran on the medium database, which has 200 atomic parts per composite part.

The OO7 benchmark defines several database traversals; these perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree. Traversals T1 and T6 are read-only; T1 performs a depth-first traversal of the entire composite part graph, while T6 reads only its root atomic part. Traversals T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts. In general, some traversals will match the database clustering well while others will not, and we believe that on average, one cannot expect traversals to use a large fraction of each page.

The OO7 database clustering matches traversal T6 poorly but matches traversals T1, T2a and T2b well; our results show that on average T6 uses only 3% of each page whereas the other traversals use 49%.

The objects in the databases in both systems are clustered into 4 KB. The database takes up 38 MB in our implementation, and it was accessed by a single client. In Thor, each OR had a 20 MB cache (of which 16 MB were used for the MOB); the FE cache had 16MB. All the results we report are for hot traversals: we preload the caches by running a traversal twice and timing the second run.

This database size is quite small for current standards. We tried to make up for that fact by maintaining small caches both at the client and server side, so that the entire database did not fit in their caches.

Overhead Performance Results

The first set of results we show are for read-only traversals. We measured elapsed times for T1 and T6 traversals of the database, both in the original implementation and the version that is replicated using the BASE technique, BASE-Thor.

The results in Figure 6-1 indicate that BASE introduces only a reasonable amount of overhead for read-only traversals. Relative to the original implementation, BASE takes 7% more time to complete the T1 traversal, and 44% more time to complete the T6 traversal. The reason the percentage overhead is greater in T6 is because the commit operation represents a larger part of the time elapsed in T6. Since the commit operation involves a lot of communication between the FE and the OR, T6 is relatively more affected by BASE than T1.

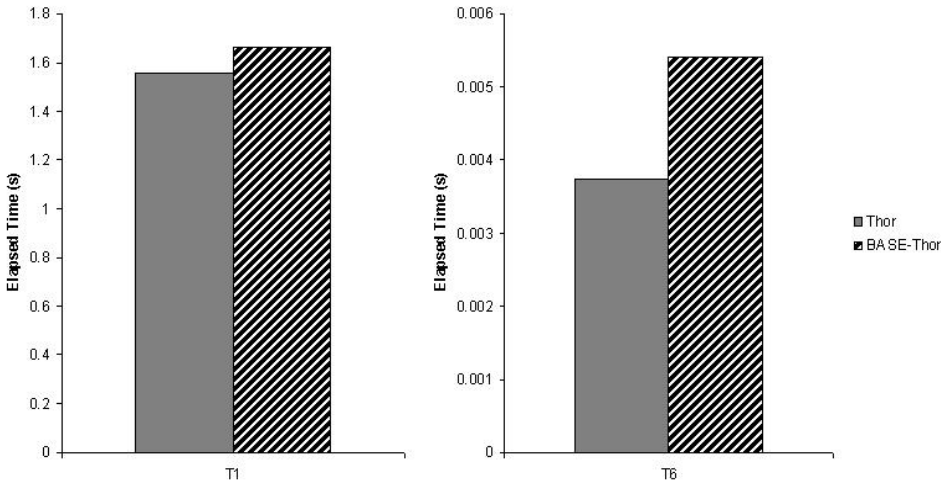

Figure 6-1: Elapsed time, hot read-only traversals

Figure 6-2 shows elapsed times for read-write traversals. In this case, BASE adds an overhead relative to the original implementation of 12% in T2a and 50% in T2b. The difference in these relative overheads is also due to distinct relative weight of the commit operation.

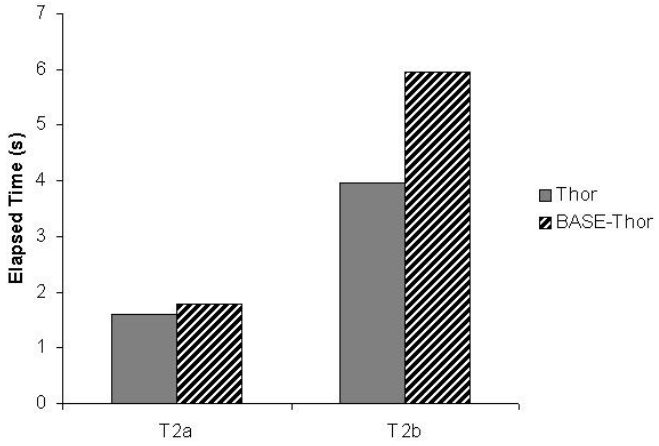

Figure 6-2: Elapsed time, hot read-write traversals

Now, we will try to explain the sources of this overhead. Figure 6-3 shows a breakdown of the time spent in a T2b traversal.

The elapsed time is divided in the time spent in the FE for the traversal (23%) and commit time (77%).

The commit time is further divided into the execution of the request (i.e. updating the Thor state such as the validation queue, MOB, etc.) which takes 10% of the time, checkpointing, which takes

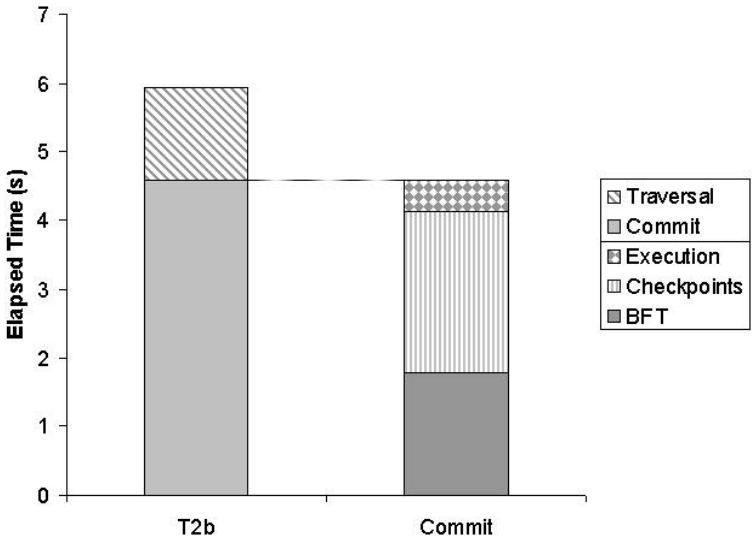

Figure 6-3: Breakdown of T2b traversal

51%, and the remaining 39% are essentially overhead introduced by the replication protocol.

Figure 6-4 shows a breakdown for the cost of checkpointing. This is dominated by calls to the abstraction function of objects that correspond to OR pages. This consists of fetching the corresponding page, which takes 65% of the total checkpointing time, and copying it to the newly created object, which is 10% of the total time. The remaining 25% of the checkpointing time consist of invoking the abstraction function for other objects and computing digests to form the state partition tree.

Fetching a page from the OR consists essentially of cache lookup and management operations (21%), reading pages from disk (53%), and installing pending modifications (26%).

We can conclude from these results that checkpointing is responsible for most of the penalty introduced by the BASE technique in this example. Most of the work during checkpoints takes place in fetching OR pages from disk.

Therefore, the fact that we use a small cache for OR pages (only 4 MB) accounts for a substantial overhead in checkpointing. A larger cache in the OR would help alleviate the overhead introduced by the BASE technique.

We mentioned before that the small cache size was used to prevent the entire database from fitting into memory. So the question is what would happen if we had a larger cache and a larger database. We believe that the performance of checkpointing would improve, since we would not need to fetch most of the pages from disk during a checkpoint, because these were changed during

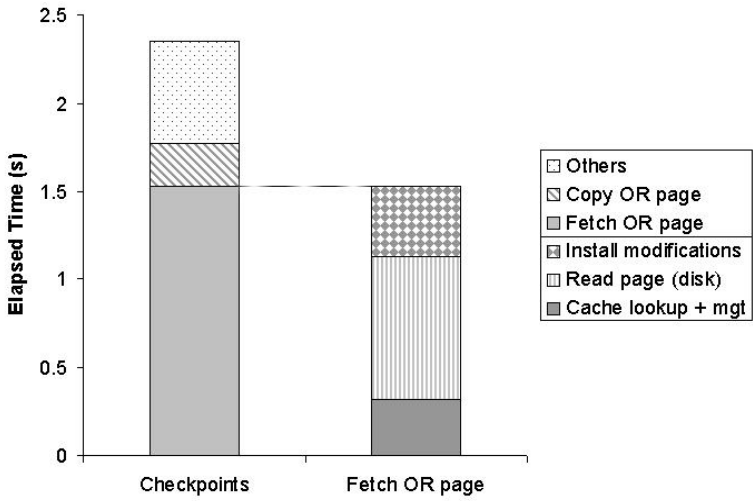

Figure 6-4: Breakdown of checkpointing

the last checkpoint interval and therefore are likely to be cached. Furthermore, the fact that we are increasing the database size should not generate cache contention during a checkpoint interval, since the number of pages that are changed in a checkpoint interval is limited by the number of operations in the interval, which in our case was 128.

6.2.2 Code Complexity

We perform here a code complexity analysis similar to the one in Section 6.1.2.

The code has a total of 658 semicolons with 345 in the conformance wrapper and 313 in the state conversion functions. To put these numbers in perspective, the number of semicolons in the original Thor code is 37055.

Furthermore, this analysis excludes the operating system code that implements all the system calls invoked by the original Thor code.

Chapter 7

Related Work

This is, to the best of our knowledge, the first work to combine abstraction and Byzantine fault-tolerance to allow the reuse of off-the-shelf implementations in a replicated system. In this chapter, we will present previous work on software fault-tolerance that closely relates to ours.

Section 7.1 discusses software rejuvenation and N-version programming is discussed in Section 7.2.

7.1 Software Rejuvenation

When a software application runs continuously for long periods of time the processes that execute it start to age or slowly degrade with respect to effective use of their system resources. The causes of the *process aging* are memory leaks, unreleased file locks, file descriptor leaks, data corruption, etc. Process aging affects the performance of the application and may lead it to fail.

Resource leaks and other problems causing processes to age are due to bugs in the application program, in the libraries it uses, or in the application execution environment (e.g. the operating system). Identifying and fixing bugs from such diverse sources can be difficult or even impossible.

Software rejuvenation [31, 22] is the concept of periodically and preemptively terminating an application, and restarting it at a clean internal state to prevent failures (caused by process aging) in the future.

The solution proposed through software rejuvenation is only preventative: it is not intended to allow programs to recover from failures but only to avoid them. Therefore it does not replace other failure recovery mechanism but complements them by reducing the likelihood that such failures will occur.

We complement the prophylactic action of periodically restarting processes with state checking and correction that is done both after the process restarts and periodically, during checkpoints.

In the original software rejuvenation approach, the server in a client-server application will be unavailable during rejuvenation, which increases the downtime of the application. Although these are planned and scheduled downtimes, which allows them to have a less disruptive effect on the system, the fact that the system is unavailable while it is being rejuvenated is a problem: to reduce downtime, rejuvenation should be done infrequently, but to reduce failures due to software aging, rejuvenation should be done often.

Our technique for software rejuvenation is based on the proactive recovery technique implemented in BFT [17]. This solves the problem of additional downtime since it maintains availability during the recovery period by having the replicas that are not recovering executing the requests. Furthermore, it allows frequent recovery without undermining the performance of the system.

In order to implement a graceful shutdown and restart when rejuvenating a process, it may be necessary to checkpoint the internal data structures of the application and restore them during restart. The original scheme does not impose any restrictions on what these checkpoints should be, so it is entirely up to the application to clean up its state during shutdown so it can continue executing after restart.

BFT allows frequent and complete checkpoints of the service state. It clearly defines the granularity of those checkpoints, so that only the incorrect part of the state needs to be fetched from other replicas. BASE improves on BFT since our use of abstraction allows us to fix an incorrect state using the states of replicas with different implementations, and it also allows us to tolerate software errors due to aging that could not be tolerated in BFT, e.g., resource leaks in the service code.

7.2 N-Version Programming

N-Version Programming [20] exploits design diversity to reduce common mode failures. It works as follows: N software development teams produce different implementations of the same service specification for the same customer; the different implementations are then run in parallel; and voting is used to produce a common result. This technique has been criticized for several reasons [25]: it increases development and maintenance costs by a factor of N or more, adds unacceptable time delays to the implementation, and it does not provide efficient state checking and recovery mechanisms. In general, this is considered to be a powerful technique, but with limited usability since

only a small subset of applications can afford its cost.

BASE enables low cost N-version programming by reusing existing implementations from different vendors. Since each implementation is developed for a large number of customers, there are significant economies of scale that keep the development, testing, and maintenance costs per customer low. Additionally, the cost of writing the conformance wrappers and state conversion functions is kept low by taking advantage of existing interoperability standards. The end result is that our technique will cost less and may actually be more effective at reducing common mode failures because competitive pressures will keep the implementations of different vendors independent.

Recovery of faulty versions has been addressed in the context of N-Version Programming, but, to the best of our knowledge, the approaches that were developed suffered from two problems. First, they are inefficient and cannot scale to services with large state. Second, they require detailed knowledge of each implementation, which make the reuse of existing implementations difficult.

Community error recovery [55] is a method for recovering faulty versions based on two levels of recovery: more frequent cross-check points where program variables in an application-chosen subset are compared and corrected if needed; and less frequent recovery points where the entire program state is compared and recovered. This approach is clearly inefficient since cross-check points only provide partial recovery and recovery points impose a large overhead (which forces them to be infrequent) and imply the transfer of a prohibitively large amount of data for most services (e.g., in a file system it must transfer all the files and directories). To compare the state from different implementations, this work proposes mappings for transferring internal states, which are similar to our mappings except that there is only one mapping from the whole service state to the common representation and another in the opposite direction.

Our technique improves on this by comparing the entire state frequently during checkpoints and after proactively recovering a replica. The state checking and transfer mechanism is efficient: it is done at the granularity of objects in the abstract state array and the state digest hierarchy allows a replica to compute a digest of the entire state and determine which objects are corrupt very efficiently, and only transfer and restore the state of those objects.

Furthermore, the abstract state is based on what is common across the implementations of the different versions and the conversion functions have glass-box access to each implementation. The BASE technique allows us to treat each implementation as a black box — the state conversion functions use only existing interfaces, which is important to allow the reuse of existing implementations. Furthermore, we derive the abstract state from an abstract behavioral specification that captures what

is visible to the client succinctly; this leads to better fault tolerance and efficiency.

In a recent work [52] on N-version programming, Romanovsky proposes to apply N-version programming to develop classes and objects in object-oriented systems. This way, when applying the community error recovery techniques, there would exist one mapping per class, instead of a mapping for the whole service state. But this approach only changes the unit of diversity, and does not solve the state comparison and state transfer problems mentioned above.

Chapter 8

Conclusions

In this thesis, we have presented a technique to build replicated services that allows the use of nondeterministic or distinct implementations without modifications. In this chapter, we summarize our work and also present areas for future research.

8.1 Summary

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks to gain control or deny access to important services. Byzantine fault-tolerance allows replicated systems to mask some software errors, but it has been expensive to deploy.

This thesis describes a novel replication technique that combines work on Byzantine fault-tolerance and abstract data types to reduce the cost of deploying Byzantine fault-tolerance and improve its ability to mask software errors.

This technique reduces cost because it enables reuse of off-the-shelf service implementations without modifications. It improves resilience to software errors by allowing efficient state comparison and correction of the state using copies of the abstract state stored in replicas that run distinct implementations, opportunistic N-version programming, and software rejuvenation through proactive recovery.

The reuse of existing implementations without modifying them and the recovery of faulty replicas in a replicated system where different replicas run different implementations of the service code are possible since abstraction hides implementation details that cause the concrete state or the behavior of different or nondeterministic replicas to diverge.

Opportunistic N-version programming runs distinct, off-the-shelf implementations at each repli-

ca to reduce the probability of common mode failures. To apply this technique, we must define a common abstract specification for the behavior and state of the service, implement appropriate conversion functions for the request and replies of each implementation in order to make them behave according to the common specification, and implement state conversions to allow state transfer between implementations with distinct concrete representations for the service state. These tasks are greatly simplified by basing the common specifications on standards for the interoperability of software from different vendors; these standards are increasingly common, e.g., ODBC [23], NFS [2], and POSIX [3]. Opportunistic N-version programming improves on previous N-version programming techniques by avoiding the high development, testing, and maintenance costs without compromising the quality of individual versions.

Proactive recovery allows the system to remain available provided no more than $1/3$ of the replicas become faulty and corrupt the abstract state (in a correlated way) within a window of vulnerability. Abstraction may enable more than $1/3$ of the replicas to be faulty because it can hide corrupt portions of the concrete state of faulty replicas. Our experimental results indicate that recovery can be performed frequently to reduce the size of the window of vulnerability and improve fault tolerance with little impact on service performance.

The methodology to build replicated systems by reusing off-the-shelf service implementations is supported by a library, BASE, that is also presented here. BASE is an extension to BFT, a Byzantine fault-tolerance library with good performance and strong correctness guarantees. We implemented BASE as a generic program library with a simple interface. We were able to add an abstraction layer to the BFT library without sacrificing performance or correctness guarantees.

The thesis described a replicated NFS file system implemented using our technique, where replicas ran different operating systems and file system implementations. The conformance wrapper and the state conversion functions in our prototype are simple, which suggests that they are unlikely to introduce new bugs and that the monetary cost of using our technique would be low.

We ran the Andrew benchmark to compare the performance of our replicated file system and the off-the-shelf implementations that it reuses. Our performance results indicate that the overhead introduced by our technique is low; it is usually around 30% for this benchmark. We have also shown an implementation where the system is running a different operating system and a different implementation of the NFS service in each one of the four replicas.

We have also used the methodology to implement a Byzantine fault-tolerant version of the Thor object-oriented database and made similar observations. We used the OO7 benchmark to measure

the performance of this system. In this case, the methodology enabled reuse of the existing database code, which is non-deterministic.

8.2 Future Work

The current prototype for the replicated version of the Thor object-oriented database does not include important aspects such as proactive recovery, support for transactions that use objects from more than one server, or garbage collection. We would like to implement these features using the design discussed in Section 5.5.

Also we would like to obtain a more complete performance evaluation of our work, namely by repeating the experiments with a heterogeneous setup in a better environment (with all the machines in the same network) and we would like to experiment measuring the performance of the system in a situation where the state is corrupted and correct state has to be fetched after proactively recovering the replica.

An important direction for future work is to apply the methodology to other services. There are two services that seem of particular relevance. The first is a relational database. Applying the methodology to a relational database is a challenging problem because the state is much more complex than the state of the applications we studied and the standard used to access that state (ODBC) is also more complex and includes certain interfaces that are difficult to translate, such as operations to discover which functions the underlying database management system implements [23]. The second service that we would like to apply our technique to is version 4 of the NFS protocol [4]. This version of the NFS protocol is more complex than the version we present in this thesis, and it would be useful to assess the amount of increased complexity in the design of the wrapper and mappings for this version.

We would also like to measure the effectiveness of opportunistic N-version programming in tolerating software errors. The way we intend to do this is by running fault injection experiments. Ideally, we would like to collect from bug logs all the bugs that were discovered (until the present date) in the four implementations of NFS that we used in our replicated file system. Then, we could reproduce them and evaluate the availability of our replicated system under different loads. Unfortunately, bug logs are not available for all of these implementations. Debian [21] provides a bug tracking system for the GNU/Linux distribution (this includes the Linux implementation of NFS). This system records details of bugs reported by users and developers. This information could

be used to reproduce actual bugs in this system, but similar information does not seem to be available for other implementations.

An alternative approach to achieve this goal is to inject random faults. Doing this enables us to measure the availability of the system as a function of the failure probability at each node and the correlation between the failures at distinct nodes. Therefore, we must add a fault injection module to the conformance wrapper that, with a certain probability $p_{failure}$ injects a random fault in the file system operation (e.g., changing the value of an argument or a reply) and with another probability $p_{correlation}$ replicates that failure in more than one replica. It is important to measure these two parameters so that we can see how effective this technique is both when software errors occur independently at distinct implementations and when different design teams produce the same errors.

In Section 3.2.2 we mentioned a shortcoming of the BASE library, namely the need to fetch a large object from other replicas, even when only a small part of that object needs to be updated. We proposed a solution that was based on changing the abstract state specification in Section 4.5. But this approach represents a conflict between a simple abstract representation and reasonably small objects. To avoid this conflict, we can include more efficient support for large objects in the BASE library. This could be done by applying the hierarchical state partitioning scheme to large objects automatically, i.e., when an object grows larger than a certain size the library divides it in fixed-size blocks and computes digests of those blocks individually. The digest of the object can be obtained incrementally from the block digests the same way that it is done for the non-leaf nodes in the partition tree. This way, fetches can be done at the granularity of blocks and not objects, whenever the object size is larger than the block size.

To make the task of writing state conversions easier, we would like to develop a library of mappings between abstract and concrete states for common data structures. This would further simplify our technique.

Bibliography

[1] Network working group request for comments: 1014. XDR: External data representation standard, June 1987.

[2] Network working group request for comments: 1094. NFS: Network file system protocol specification, March 1989.

[3] IEEE std 1003.1-1990, information technology Portable Operating System Interface (POSIX) part 1: System application program interface (API) [C language]. IEEE, New York, 1990.

[4] Network working group request for comments: 2624. NFS version 4 design considerations, June 1999.

[5] A. Adya. Transaction Management for Mobile Objects Using Optimistic Concurrency Control. Master's thesis, Massachusetts Institute of Technology, January 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-626.

[6] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[7] S. Ahmed. A scalable byzantine fault tolerant secure domain name system. Master's thesis, Massachusetts Institute of Technology, 2001.

[8] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology – EUROCRYPT' 97*, 1997.

[9] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO'99*, pages 216–233, 1999.

[10] B. Callaghan. *NFS Illustrated*. Addison-Wesley, 1999.

[11] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-Graned Sharing in a Page Server OODBMS. In *Proceedings of the 1994 ACM SIGMOD*, Minneapolis, MN, May 1994.

[12] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[13] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2000.

[14] M. Castro, A. Adya, B. Liskov, and A. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 102–115, St. Malo, France, October 1997.

[15] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[16] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[17] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[18] M. Castro, R. Rodrigues, and B. Liskov. Using abstraction to improve fault tolerance. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[19] CERT Analysis Center. http://www.cert.org/, 2001.

[20] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing, FTCS-8*, pages 3–9, 1978.

[21] Debian bug tracking system. http://www.debian.org/Bugs/, 2001.

[22] S. Garg, Y. Huang, C. Kintala, and K. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *ACM SIGMETRICS Conference on measurement and modeling of compuer systems*, pages 252–261, May 1996.

[23] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.

[24] S. Ghemawat. *The Modified Object Buffer: a Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, 1995. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-656.

[25] J. Gray and D. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, September 1991.

[26] J. N. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[27] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[28] T. Haerder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2):111–120, June 1984.

[29] M. P. Herlihy and J. M. Wing. Axioms for Concurrent Objects. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[30] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[31] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, modules and applications. In *Fault-Tolerant Computing, FTCS-25*, pages pages 381–390, Pasadena, CA, June 1995.

[32] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. of the Hawaii International Conference on System Sciences*, Hawaii, January 1998.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[34] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[35] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, Portugal, June 1999.

[36] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, San Mateo, California, USA, 1993. Also published as Programming Methodology Group Memo 77, MIT LCS.

[37] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 226–238. ACM Press, 1991.

[38] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[39] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[40] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *Third International Conference on Parallel and Distributed Information Systems, Austin*, September 1994.

[41] U. Maheshwari and B. Liskov. Collecting Cyclic Distributed Garbage by Controlled Migration. *Distributed Computing*, 10(2):79–86, 1997.

[42] U. Maheshwari and B. Liskov. Partitioned Collection of a Large Object Store. In *Proc. of SIGMOD International Conference on Management of Data*, pages 313–323, Tucson, Arizona, May 1997. ACM Press.

[43] U. Maheswari and B. Liskov. Collecting Cyclic Distributed Garbage using Back Tracing. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, August 1997.

[44] D. L. Mills. Network time protocol (version 1) specification and implementation. DARPA-Internet Report RFC 1059, July 1988.

[45] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 76–88. ACM, August 1983.

[46] J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proc. of USENIX Summer Conference*, pages 247–256, Anaheim, CA, June 1990.

[47] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[48] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[49] R. Rivest. The MD5 message-digest algorithm. Internet RFC-1321, April 1992.

[50] R. Rodrigues, M. Castro, and B. Liskov. BASE : Using abstraction to improve fault tolerance. In preparation.

[51] R. Rodrigues, K. Jamieson, and M. Castro. A Liveness Proof for a Practical Byzantine Fault-Tolerant Replication Algorithm. In preparation.

[52] A. Romanovsky. Faulty version recovery in object-oriented N-version programming. *IEE Proceedings - Softw.*, 147(3):81–90, June 2000.

[53] R. Sandberg et al. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, June 1985.

[54] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[55] K. Tso and A. Avizienis. Community error recovery in N-version software: A design study with experimentation. In *Fault-Tolerant Computing, FTCS-17*, pages 127–133, Pittsburgh, PA, July 1987.