Chapter 4

A Guided Tour on the Theory and Practice of State Machine Replication

Alysson Neves Bessani (University of Lisbon, Faculty of Sciences. Portugal)
Eduardo Alchieri (University of Brasília, Dep. of Computer Science. Brazil)

Abstract

This chapter presents the fundamentals and applications of the State Machine Replication (SMR) technique for implementing consistent fault-tolerant services. Our focus here is threefold. First we present some fundamentals about distributed computing and three "practical" SMR protocols for different fault models. Second, we discuss some recent work aiming to improve the performance, modularity and robustness of SMR protocols. Finally, we present some prominent applications for SMR and an example of the real code needed for implementing a dependable service using the BFT-SMART *replication library.*

4.1. Introduction

Service replication is a technique in which copies of the service are deployed in a set of servers instead of in just one. This technique is often used with two objectives: increase the system performance and capacity or provide fault tolerance. Replication can increase the service' performance and capacity since the replicas provide more resources to the service. Fault tolerance can be achieved with several replicas through the use of spatial redundancy to the system, making it continue to operate despite the failure of a fraction of these replicas.

Regarding the maintenance of consistent state in replicated systems, there are two fundamental approaches: primary-backup (or passive) replication (Alsberg and Day 1976) and state machine (or active) replication (Lamport 1978a). In the primary-backup replication model there is one primary replica that executes all operations issued by the clients and, periodically, pushes state updates to the backup replicas. Furthermore, these replicas keep monitoring the primary to ensure that one of them takes over its role in case it fails. In the state machine replication model clients issue commands to all replicas, which execute them in a coordinated way and produce a reply. An important advantage

of this model is that no monitoring or synchronization is required (at least at the service level), since all replicas are kept synchronized.

In this chapter we present a guided tour to the most important results regarding the theory and practice of *State Machine Replication (SMR) for fault tolerance*. During the next sections we discuss some of the core SMR protocols together with more recent works in the field and present the main applications for this technique.

A bit of history. The classical "logical clocks" paper by Lamport introduced the notion of state machine replication (Lamport 1978b), however, the algorithm described in that paper was not fault-tolerant. The first fault-tolerant state machine replication algorithm was described in the same year (Lamport 1978a). During the eighties and nighties much effort has been put on broadcast protocols (Schiper et al. 1991; Hadzilacos and Toueg 1994) and group communication systems (Chockler et al. 2001; Powel 1996), which can also be used to implement replicated state machines. The fundamental text about state machine replication is a tutorial by Fred Schneider (Schneider 1990), where the approach is fully specified without the non-fundamental details about the protocols required for implementing it. The most well-known state machine replication algorithm is the Paxos protocol (Lamport 1998). This algorithm, devised in 1989, is quite similar to the Viewstamped Replication protocol, devised approximately at the same time for database replication (Oki and Liskov 1988).

In 1999 the first practical Byzantine Fault-Tolerant (BFT) protocol was published (Castro and Liskov 1999; Castro and Liskov 2002). The key innovation of this algorithm is the avoidance of public key signatures, which leads it to achieve a performance similar to non-replicated systems. PBFT started a great interested in BFT replication during the following decade (Yin et al. 2003; Abd-El-Malek et al. 2005; Cowling et al. 2006; Kapitza et al. 2012; Kotla et al. 2009; Guerraoui et al. 2010), but with little practical adoption up to this point.

In the last years, the rising of internet-scale services requiring fault tolerance at their core, renewed the interest in state machine replication (Chandra et al. 2007). Modern datacenter systems are designed for faults, and SMR can be used for implementing some fault-tolerant core services in which much larger systems can rely upon. For example, Paxos and similar algorithms are being extensively used for implementing storage (Bessani et al. 2013; Bolosky et al. 2011) and coordination (Burrows 2006; Hunt et al. 2010) systems deployed in these infrastructures.

Chapter Organization. This chapter is organized in the following way. Section 4.2 presents the basic concepts about SMR, discussing some fundamental results in distributed computing. Section 4.3 describes the three most fundamental protocols for implementing SMR under three important fault models. Section 4.4 discusses some recent work and extensions proposed to the baseline SMR protocols. Section 4.5 shows some applications for SMR. Section 4.6 describes a practical library for implementing SMR applications and shows an example of a service developed using it. Section 4.7 presents our final remarks.

Through this paper we discuss and present the work we consider most relevant for developers and researchers interested in devising highly-available services. Naturally, this choice is highly subjective and by no means exhaustive. There are many other interesting works and systems either directly related with SMR or in related fields, such as agreement and broadcast algorithms, database replication, weakly consistent replica maintenance or datacenter infrastructures, that are also relevant for the reader interested in the area.

4.2. Basic Concepts

This section presents the definition of state machine replication, discussing aspects such as the system model assumptions and some distributed computing fundamental results that impact the state machine replication protocols.

4.2.1. State Machine Replication: Definition and Properties

State machine replication is usually defined through a set of clients submitting commands to a set of replicas behaving as a "centralized" service, or a *replicated state machine* (RSM). The implementation of such paradigm requires three properties to hold (Schneider 1990):

- *Initial state:* All correct[1] replicas start on the same state;
- *Determinism:* All correct replicas receiving the same input on the same state produce the same output and resulting state.
- *Coordination:* All correct replicas process the same sequence of commands;

In principle, one could argue that the first two requirements are trivial and that the crux of the problem in implementing RSMs is to provide Coordination. This is reflected by the fact that most works in the field aims to provide efficient replica coordination protocols. These protocols implement on its core a *total order multicast* and/or a *consensus* algorithm (Hadzilacos and Toueg 1994). However, from a more practical point of view, Initial State and Determinism are not so simple to ensure. The complexity in satisfying the former appear in the practical scenarios where replicas crash (or are turned off for maintenance) and later recover, with a state that is (possibly) outdated when compared with the other replicas. Determinism, on the other hand, can severely constraint the service performance, since it complicates the use of multiple threads and cores (Kapritsos et al. 2012).

Assuming these three requirements are satisfied by a RSM implementation, the system must satisfy the following safety and liveness properties:

- *Safety:* all correct replicas execute the same sequence of operations;
- *Liveness:* all correct clients operations are executed.

[1]To be general enough to accommodate BFT protocols, we consider properties ensured only for correct replicas, since usually nothing can be said about faulty processes under the Byzantine fault model.

The Safety property allows the implementation of strongly consistent services, satisfying the consistency property known as *Linearizability* (Herlihy and Wing 1990). Although there is a vast literature about weakly-consistent replication (e.g., the CAP theorem and its tradeoffs), in this tutorial we focus only on traditional SMR, where Consistency (Safety) is always ensured.

4.2.2. Programming a RSM

Conceptually, the basic programming model of a RSM comprises a set of clients invoking RPC-like operations to a deterministic service implemented by the system, as illustrated in Figure 4.1.

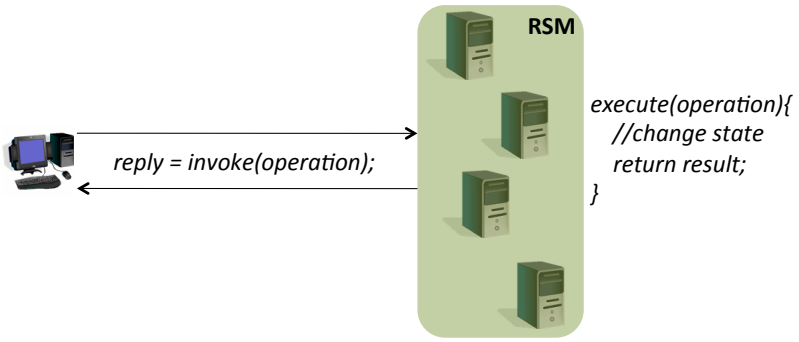

Figure 4.1. The basic programming model of a replicated state machine.

This simple model hides two fundamental limitations. First, in order to satisfy the determinism requirement, the service supported by the replicas is, in principle, single threaded[2]. This limitation is specially relevant when considering the large number of cores present in modern servers. Second, the RSM cannot initiate communication with clients, being thus limited to answer requests. Consequently, clients need to keep polling the RSM to discover state changes, which can generate additional load and degrade the system performance. As will be seen in Section 4.5, practical systems usually break the modularity between the SMR protocol and the service being implemented to cope with this limitation.

Besides the usual *invoke* operation, used for submitting commands to the replicated state machine, it is common for replication libraries to support an additional operation for invoking read-only operations in the RSM. This separation between read-only and read-write operations is important because in most SMR protocols the former can be processed (in principle) without coordination, since the state of the service is not changed.

At the server side, the service code should implement at least four callbacks to (1) execute an operation, (2) execute a read-only operation, (3) serialize a snapshot of the service state, and (4) deserialize and update the service state.

[2]There are some solutions for running multithread servers in replicated state machines to ensure the state evolves deterministically (Kapitza et al. 2010; Kapritsos et al. 2012), but they tend to be conceptually complex when compared with this basic design.

4.2.3. System Models

SMR protocols are usually defined in terms of a fully connected distributed system in which each pair of processes communicates through bidirectional channels. This model is complemented by a set of assumptions defining the failures that can happen in the system, its synchrony and the cryptographic primitives that are available.

4.2.3.1. Fault Models

One of the key objectives in implementing a service as a RSM is to tolerate faults. Normally, two kinds of faults are considered for the processes (clients and servers) in a SMR-based system:

- *Fail-stop*: A faulty (crashed) process halts and does not execute any further operations during the system execution;
- *Byzantine*: A faulty process may behave arbitrarily, i.e., deviating from its algorithm and taking any action.

Most practical systems consider extensions of these models in which a faulty process can be recovered and resume its normal execution.

It is also important to define which kind of communication channels errors can affect the system. The main fault models for channels are:

- *Reliable*: every message sent will be delivered;
- *Unreliable*: messages can be modified, generated or dropped;
- *Lossy*: messages may be lost, but not modified;
- *Fair*: if a message is resent infinitely often, it will arrive infinitely often.

An important implicit assumption in replicated systems is that correlated failures will not happen, i.e., that the probability of a replica to fail is statistically independent from the probability of any other replica of the system to fail (Littlewood et al. 2001). In practice, this property can be enforced by deploying replicas in different platforms and locations, relying to different power sources and network infrastructures and by design diversity through the use of N-version programming (Obelheiro et al. 2006). Furthermore, when malicious Byzantine faults are considered it is important to apply software diversity for minimizing the chances of a common bug or vulnerability bring the whole system down (Garcia et al. 2013; Junqueira et al. 2005).

4.2.3.2. Synchrony

Synchrony assumptions are mostly related with the timing aspects of the system. There are several synchrony models (Attiya and Welch 2004; Lynch 1996), and in this section we briefly discuss the three of them that are relevant for discussing practical SMR protocols.

The weakest synchrony model is the *asynchronous* distributed system (Fischer et al. 1985; Lynch 1996). In this model, processing and communication have no time bounds, and there are no physical clocks. Therefore, in this model the notion of time does not even exists.

On the other end of the synchrony spectrum there are the *synchronous* distributed systems (Lynch 1996). In these systems, there are known bounds in terms of processing, communication and clock error in different processes. This model represents a real-time system.

An intermediate model that is often considered in practical systems is the *partially synchronous* system model (Dwork et al. 1988). This model considers that the system behaves asynchronously until a certain (unknown) instant *GST* (Global Stabilization Time), in which the system becomes synchronous, respecting some (unknown) processing and communication time bounds. In practice, the system does not need to be synchronous forever, but only during the execution of the protocol of interest.

The relevance of the partially synchronous model is that it models the expected behavior of best-effort networks such as the internet. In these systems, the network is expected to present a stable behavior (i.e., acting as a synchronous system), but sometimes can be subject to perturbations that make its behavior unpredictable as in an asynchronous system.

4.2.3.3. Cryptography

Byzantine fault-tolerant SMR protocols usually consider the existence of authenticated communication channels, which can only be implemented if either a Public-key infrastructure is available for supporting the use of asymmetric cryptography for message signatures or the existence of shared secrets between each pair of processes (which can be supported by a key distribution center like Kerberos) for enabling the use of Message Authentication Codes (MAC) (Bishop 2002). In both cases, it is always assumed that there exists a collision-resistant hash function for implementing these primitives (e.g., SHA-1) and, additionally, a symmetric (e.g., AES) or asymmetric (e.g., RSA) cryptographic algorithm (Goldereich 2001).

4.2.4. Some Fundamental Results in Distributed Computing

In this section we present five fundamental results from distributed computing theory that constrains the design space of practical state machine replication protocols.

(R1) Impossibility of reliable communication on top of unreliable channels. One assumption that simplifies the implementation of distributed algorithms in the message passing model is the existence of reliable channels, as defined in the previous section. A reliable channel $c_{p,q}$ allows a process p to put messages (e.g., through the $send(m)$ primitive) that will eventually be delivered to a destination process q (e.g., through the $receive(m)$ primitive).

One fundamental question is how to implement reliable channels in a best-effort network such as the internet. In principle, one may argue that TCP provides such abstraction. However, the fact that a TCP connection may be broken even if two communicating processes are correct implies this is not the case. In the end, TCP is built over a best-effort protocol (IP), which gives no reliability guarantees.

A fundamental result related with the *two generals problem* (Akkoyunlu et al. 1975) shows that it is impossible to provide reliable communication on top of unreliable (i.e., lossy) channels. However, a quite weak assumption about the reliability of the channel (fair link, described in previous section) makes it possible to implement such desired property.

In a nutshell, a reliable channel can be implemented on top of a fair channel through the use of retransmissions and confirmations. More precisely, the sender keeps sending a message m periodically until it receives an ACK from its destination process, which will send this confirmation every time it receives m (even if the message is delivered just once to the upper layers). Additionally, authentication and integrity protection can be implemented through the use of MACs on the exchanged messages.

(R2) Equivalence between total order and consensus. As already discussed, a fundamental requirement for implementing SMR is the coordination of replicas, which requires that all correct replicas receive the same messages in the same order. Conceptually, satisfying this requirement requires the implementation of a total order multicast (or atomic multicast) protocol. A fundamental result in distributed computing is the equivalence between the problem of implementing this kind of communication primitive and solving consensus (Hadzilacos and Toueg 1994). In the consensus problem, processes propose some value and try to reach agreement about one value to decide, which must be one of the proposed values.

This equivalence is important because it shows that the implementation of a RSM is subject to the same constraints and results of the well-studied consensus problem, which leads us to our next fundamental result.

(R3) Impossibility of fault-tolerant consensus in asynchronous systems. One of the most well-known results in distributed computing is the impossibility of achieving (deterministic) consensus in an asynchronous system in which a single process can crash (Fischer et al. 1985). This result is sometimes called the *FLP impossibility*.

Intuitively, this impossibility comes from the difficulty in identifying crashed processes (i.e., detecting faults) in asynchronous systems. More precisely, during the execution of a consensus protocol, if there is a "step" in which a process p is waiting for a

message from process q in order to decide what to do in the next "step" of the protocol, p does not know if q crashed or is very slow until the message arrives. If the message does not arrive, p have to decide if it (1) waits more or (2) assumes q is faulty and proceed without the message. The FLP impossibility shows, using sophisticated proof techniques, that this problem (1) can make the consensus protocol never terminate or (2) can lead different processes to decide different values. Making thus impossible to devise a consensus protocol that satisfies both Termination and Agreement (Fischer et al. 1985).

The impossibility of fault-tolerant consensus in asynchronous systems had a huge impact both in the theory and practice of replication protocols, opening new avenues for research in ways to circumvent this result in practical system models.

(R4) Minimum synchrony required for fault-tolerant consensus. After the impossibility of fault-tolerant consensus was published, several researchers tried to circumvent it. In a seminal work by Dwork, Lynch and Stockmeyer (Dwork et al. 1988) the concept of *partial synchrony* was defined as an intermediate model between synchronous and asynchronous systems, and devised algorithms with optimal resilience in this model.

One of the fundamental contributions of this work is to show that by separating the safety and liveness properties of the protocol, it is possible to solve consensus in a *partially synchronous* system model as described in Section 4.2.3. The separation of liveness and safety was later exploited in all practical agreement protocols (e.g., (Lamport 1998; Castro and Liskov 1999)), which ensure that there will be no safety violations (e.g., non-unanimous decisions that lead to conflicts in message delivery ordering) during the asynchrony periods, but can only terminate if the system behaves synchronously.

(R5) Fault thresholds. Different system models require a different number of processes to implement consensus and total order broadcast tolerating up to f failures. Table 4.1 presents some results adapted from (Dwork et al. 1988) showing the minimal number of processors for which a f-resilient consensus protocol exists. In the table we consider the synchronous and partially synchronous system models for crash, Byzantine and authenticated Byzantine failures[3]. These results reflect the exact number of replicas a f-resilient RSM must contain in order to implement replica coordination. We complement these results with the minimal number of replicas required to execute commands after it is delivered in total order, i.e., the number of replicas that are required to execute a command to enable the client to receive a correct result despite the occurrence of up to f failures.

As can be seen in the table, when considering the practical partially synchronous system model, the number of replicas required for ordering messages is $2f+1$ in a crash-prone system (as used, for example, in the Paxos protocol (Lamport 1998)), while $3f+1$ replicas are required for tolerating Byzantine faults. However, once the total order is established, f less replicas are required to execute the operations in both fault models. In Section 4.4.2 we show how this result was used in practical SMR architectures.

[3]In the authenticated Byzantine model replicas can suffer Byzantine faults but messages are signed and thus impersonations can be easily detected (Lamport et al. 1982).

Failure type	Synchronous	Partially synchronous	Replicated Execution
Fail-stop	$f+1$	$2f+1$	$f+1$
Authenticated Byzantine	$f+1$	$3f+1$	$2f+1$
Byzantine	$3f+1$	$3f+1$	$2f+1$

Table 4.1. Fault thresholds considering consensus in two system models and replicated execution of operations.

4.3. Baseline Protocols

In this section we describe three core protocols for implementing RSM in different fault models. These protocols were chosen because they work under practical assumptions and are optimal in terms of number of replicas and number of communication steps without weakening the safety and liveness properties of SMR.

As described before, the main challenge in implementing RSMs is to ensure that the operations requested by clients are executed in the same order at all replicas in spite of concurrency and failures (Section 4.2.1). All the protocols described in this section solve this problem by relying on a leader replica (sometimes called sequencer or primary[4]) to assign the order for client requests while the other replicas verify and accept the order defined by the leader. This approach has been proved to be the most interesting in practical scenarios, since it results in a simple ordering protocol for cases when the leader is correct and the system is under a period of synchrony (this is the expected *normal case*). However, if the leader is faulty (or suspected to be faulty), these protocols allow a different replica to become the leader, which is chosen through a *view change* protocol. The election is based on the number of the view, which starts in 0 and is increment during each view change (the leader of view v is the replica $v\%n$). Moreover, a faulty replica that recovers is reintegrated in the system through a *recovery* protocol. All protocols are explained considering these three subprotocols, together with some specific optimizations. We conclude this section with some general optimizations and extensions that can be applied to more than one of these protocols.

System Model. All protocols assume a partially synchronous system model in which all replicas communicate through fair channels. There are is unbounded number of clients and n replicas with unique identifiers. A unknown number of clients and up to f replicas may be faulty. Furthermore, faulty replicas can recover and resume processing in the system.

4.3.1. Paxos and Viewstamped Replication

Most practical crash fault-tolerant replicated state machines are built around the Paxos agreement algorithm (Lamport 1998). Since the original Paxos describes only a consensus protocol (Synod) and an algorithmic framework for maintaining synchronized replicas with minimal assumptions (called Multi-Paxos), we focus here on the description of

[4]Not to be confused with the primary replica in a primary-backup system. In this case the primary is used only for ordering requests, which are executed by all replicas.

Viewstamped Replication (VR), a similar (but more concrete) protocol devised independently at the same time by Oki and Liskov (Oki and Liskov 1988; Liskov and Cowling 2012).

The relevance of Paxos/VR comes from the fact this algorithm has served as the foundation for many recent replicated (consistent and fault-tolerant) data stores, coordination and configuration management systems (e.g., Apache' Zookeeper (Hunt et al. 2010)), experimental block-based data stores or virtual discs (Lee and Thekkath 1996; Rao et al. 2011; Bolosky et al. 2011; Bessani et al. 2013), and even wide-area replicated databases, such as Google Spanner (Corbett et al. 2012). Many of these applications will be discussed later in Section 4.5. As a consequence, this protocol is considered a *de-facto* standard for implementing strongly consistent crash fault-tolerant services.

Fault Model. Paxos/VR requires $n = 2f + 1$ replicas, in which up to f can be subject to crash faults. The rationale for this threshold is the following. The system has to be able to execute a request without waiting for f replicas, since these replicas may be crashed and do not send responses. However, the f replies that are not considered might merely be from processes that are slow (e.g., due to a network congestion), and f processes that replied may subsequently fail. In order to ensure safety, the protocol must ensure that, even if these f replicas fail, there will be at least one replica that processed the request and that will participate in other protocol executions. This implies that any two quorums of replicas accessed in the protocol must intersect in at least in one correct replica. Consequently, each step of the protocol must be processed by a quorum of at least $f + 1$ replicas that, together with the other f replicas that may not reply, compose the $n = 2f + 1$ requirement.

Normal Case. Figure 4.2 illustrates the messages exchanged in Paxos/VR for an update operation (a client request that modifies the application state) when the leader is correct and the system is in a synchrony period. The protocol works as follows:

1. The client sends a *Request* with the operation to be executed to the leader replica;

2. The leader chooses a sequence number i for the received request, writes it to its log, and disseminates the request and its sequence number to all other replicas in a PREPARE message;

3. If the replicas did not assign any other request to sequence number i, they accept the leader proposal and write the update (request plus sequence number) to their log, replying with a PREPARE-OK message to the leader;

4. The leader waits for f confirmations from other replicas and then executes the request and sends the *Reply* to the client. This ensures that a majority of the replicas have the request in their logs and, consequently, the request will be visible even if the leader fails;

5. Normally, the leader informs the other replicas about this request execution in the next PREPARE message, making them also execute the client request for updating their states, without sending a reply to the client.

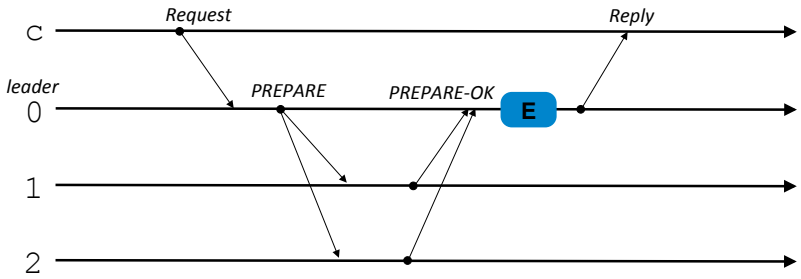

Figure 4.2. Viewstamped Replication normal case protocol.

If a client does not receive a reply for a request within a given time period, it resends the request to all replicas, ensuring that it reaches a possible new leader (see below). Client requests are ignored by non-leader replicas.

When compared with the original Paxos for implementing RSMs (Multi-Paxos) (Lamport 1998), VR presents two subtle "improvements". First, in Paxos the leader sends an explicit COMMIT message to make the other replicas *learn* that the request can be executed. Second, since Paxos explicitly considers the durability of the service (see Section 4.4.2), all accepted PREPARE messages are logged in stable storage. Figure 4.3 illustrates these two differences.

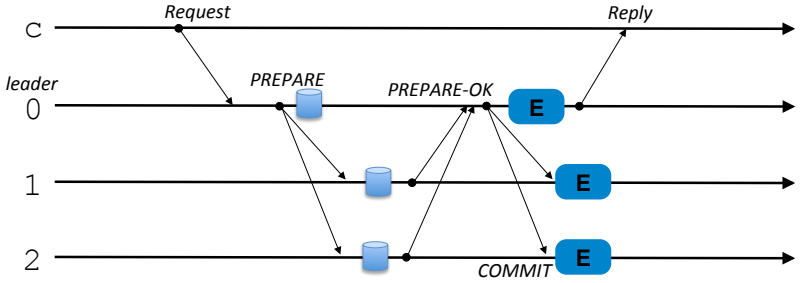

Figure 4.3. (Multi-)Paxos normal case protocol.

Optimizations. Read-only operations can be executed directly by the leader without contacting the other replicas. Furthermore, if the replicas need to learn that the client request was executed, instead of having the explicit COMMIT message as in classical Paxos, they can send the PREPARE-OK messages to all other replicas, and each replica can execute the request if it receives f of these messages (and the PREPARE from the leader). Figure 4.4 illustrates this version of the protocol.

View Change. If the leader fails, messages will not be ordered and the system will stop executing client requests. To recover from this situation, non-leader replicas monitor the

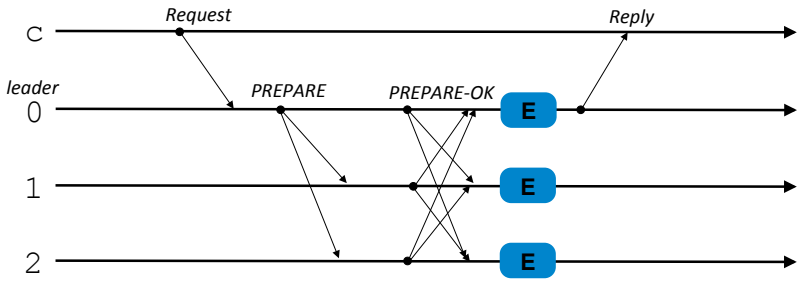

Figure 4.4. Fast execution in Viewstamped Replication.

leader and if they suspect[5] that it is faulty, a new leader is elected to ensure the algorithm makes progress. Figure 4.5 shows the messages exchanged to choose and initialize a new leader:

1. A replica that suspects the leader is faulty sends a START-VIEW-CHANGE message to all other replicas asking for a view change;

2. Each replica that receives this message sends a DO-VIEW-CHANGE message with its log to the new leader;

3. The new leader waits for $f+1$ of these messages, updates its log with the requests in all received logs, sends a START-VIEW message with its log to other replicas and starts to accept new client requests;

4. All replicas that receive the START-VIEW message synchronize their logs with the new leader and, if there exists some non-prepared request in the log, send a PREPARE-OK message to it ensuring that pendent requests will be executed.

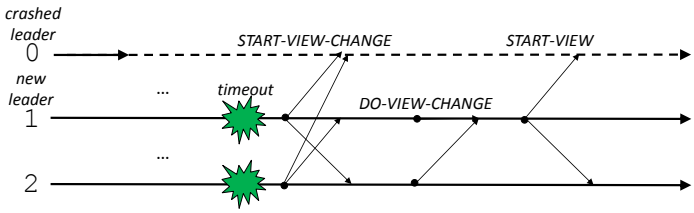

Figure 4.5. Viewstamped Replication view change protocol.

The rationale behind this protocol is that in order for a request to be executed, it must be in the log of at least $f+1$ replicas. Consequently, the order assigned for a request is preserved by the view change protocol since the new leader collects $f+1$ logs (by quorum intersection properties, at least one log contains the request).

[5]There might be false suspicious due to the asynchrony of the system.

Recovery. When a replica restarts after a crash, it cannot participate in request processing and view changes until it has a state at least as recent as when it failed. For example, if a replica forgets that it prepared some request, this information might be known to fewer than $f + 1$ replicas and, consequently, this request could be lost in a view change even if it was executed.

A simple implementation of recovery is by storing the request log in the disk before sending messages in a way that it could retrieve its state from it (the replica will not forget anything that was done before). However, this approach (which is used in Paxos) may add a non-negligible overhead in the normal case execution due to the high latency of stable storage access. VR uses the following recovery protocol that does not need disk accesses:

1. The recovering replica sends a message to all other replicas asking for the current state;
2. Each replica sends a reply with its log, among other information (Liskov and Cowling 2012);
3. The recovering replica waits for $f + 1$ of such messages,[6] including one from the current leader it discovered from the received messages. Then, it updates its state from the received leader log. At this point the replica is recovered and can resume execution.

4.3.2. PBFT and Byzantine Paxos

PBFT (Castro and Liskov 1999; Castro and Liskov 2002) is the first BFT protocol that achieved a performance similar to its crash fault-tolerant counterparts. Like Paxos and VR, it uses a combination of primary and backup replicas to order and execute requests: the primary (also called leader) assigns sequence numbers to requests, while the backups check these numbers for consistency and monitor the primary to detect when it stops or misbehaves.

Fault Model. PBFT requires $n = 3f + 1$ replicas, in which up to f can be subject to Byzantine faults. The rationale for this threshold is similar to VR, but adapted for Byzantine failures. The system has to be able to execute a request without waiting for f replicas, since these replicas may be faulty and do not send responses. However, the f replies that are not considered in the execution might merely be from processes that are slow (e.g., because of network congestion), and f processes that replied may be malicious (we can not trust on their responses). In order to ensure safety, the protocol must be executed in a way that these f responses from malicious replicas are masked. Furthermore, to ensure a processed request will be seen in case of failures, it is required that each two quorums intersect in at least $f + 1$ replicas, i.e., with at least one correct replica. Consequently, each step of the protocol must be processed by at least $2f + 1$ replicas. These $2f + 1$ replicas together with the f that may not respond define the $3f + 1$ threshold.

[6] The recovering replica is considered faulty until it finishes the recovery protocol.

Normal Case. Figure 4.6 shows the messages exchanged in PBFT for executing a client-issued operation when the leader is correct and the system is under a synchrony period.

A key feature of PBFT is the use of MAC vectors instead of digital signatures for achieving message authentication[7] (Castro and Liskov 2002). This enables the protocol to achieve a performance comparable to crash fault-tolerant protocols. The protocol works as follows:

1. The protocol begins with a client sending a request m to all replicas;
2. The leader replica then sends a PRE-PREPARE message to all replicas assigning a sequence number i to m;
3. A replica *accepts* a PRE-PREPARE message if the leader proposal is *good*, i.e., the authenticity of m is verified[8] and no other PRE-PREPARE message was accepted for the sequence number i;
4. When a replica accepts a PRE-PREPARE message, it sends a PREPARE message with m and i to all servers;
5. When a server receives $\lceil \frac{n+f}{2} \rceil$ PREPARE messages with the same m and i, it marks m as *prepared* and sends a COMMIT message with m and i to all servers;
6. When a server receives $\lceil \frac{n+f}{2} \rceil$ COMMIT messages with the same m and i, it *commits* m, i.e., accepts that request m is the i-th request to be executed;
7. Once a server executed all requests with lower sequence number, it executes m and sends a reply to the client;
8. The client waits for $f+1$ matching replies for the request and completes the operation.

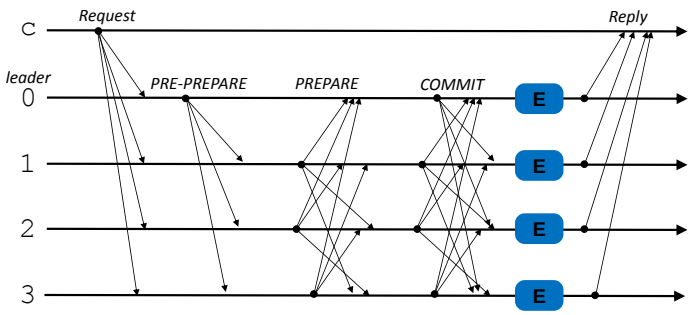

Figure 4.6. PBFT normal case protocol.

[7] If a message is sent only to one destination, it uses only one MAC instead of a vector of MACs.

[8] A replica i can authenticate m if the MAC for i in the vector of MACs of m is correct, or i has received $f+1$ PRE-PREPARE or PREPARE messages for m.

When comparing PBFT with VR/Paxos it is possible to see that besides using f additional replicas, it also requires one extra communication step. In fact, the PREPARE phase is employed for the non-leader replicas to verify if the leader made a consistent proposal in its PRE-PREPARE message. Furthermore, just like in VR/Paxos, while the PREPARE phase of the protocol ensures that there cannot be two prepared messages for the same sequence number i (two quorums of $2f+1$ replicas will intersect in at least one correct replica), the COMMIT phase ensures that a message committed with sequence number i in a correct replica will keep this sequence number even if the normal phase is interrupted by a view change.

Optimizations. Besides the use of MAC vectors instead of digital signatures, the PBFT protocol can also be made even more efficient under the assumption that failures, concurrency and asynchrony are rare in the system.

- *Read-only requests*: these requests generally do not require ordering because they do not change the replica state. All replicas can immediately reply to the client, that completes the read if it receives $2f+1$ matching replies (instead of $f+1$ - even for the ordering protocol) to ensure linearizability (Herlihy and Wing 1990). Otherwise (due to faulty replicas or concurrency), the client retransmit the request using the normal protocol.

- *Tentative execution*: replicas can tentatively execute a request when it is prepared and they have committed all requests with lower sequence number, reducing the protocol latency from 5 to 4 communication steps. The client needs to wait for $2f+1$ matching replies from different replicas to be sure that the execution order will eventually commit. If the client does not receive these replies and a timer expires, it resends the request without asking for tentative execution.

View Change. When the leader is suspected to be faulty, a new leader must be elected through the view change protocol. When a new leader is elected, it collects the protocol state from $\lceil \frac{n+f}{2} \rceil$ replicas and propagates this information to the new view. The protocol state comprises information about accepted, prepared and committed requests. This information allows the new leader to verify if some request was already committed with some sequence number. Then, the new leader continues to order requests ensuring that if a request m was already committed with sequence number i by some correct replica in the previous view, it must propose i as the sequence number for m in the new view. Consequently, even with leader failures, the safety of the protocol will still hold.

View changes are triggered by a timeout: when a non-leader replica receives a client request, it starts a timer. When the request is committed, the timer is stopped. If the timer expires before this, the replica suspects that the leader is faulty and starts a view change. The messages exchanged in PBFT for a view change, described below and represented in Figure 4.7, also use vectors of MACs for authentication[9].

[9]They correspond to the signature-free protocol from (Castro and Liskov 2002), not the original one which requires signatures on view changes (Castro and Liskov 1999).

1. When a replica suspects that the leader is faulty, it sends a VIEW-CHANGE message to all replicas. This message contains information about accepted, prepared and committed requests;

2. When a replica receives a VIEW-CHANGE message, it sends to the leader of the next view a VIEW-CHANGE-ACK message containing a digest of the received VIEW-CHANGE message;

3. The new leader only considers a VIEW-CHANGE message if it receives $2f - 1$ VIEW-CHANGE-ACK messages for it. These VIEW-CHANGE-ACK messages are used for building a certificate that proves the VIEW-CHANGE message authenticity;

4. After receiving enough information to start a new view (it may be necessary to wait VIEW-CHANGE messages from all correct replicas), the new leader starts a new view and sends a NEW-VIEW message to all other replicas. This message contains all the information necessary to start the next view;

5. Once a replica receives the NEW-VIEW message, it starts the next view if the information on the message is correct (Castro and Liskov 2002). Otherwise, the replica starts another view change.

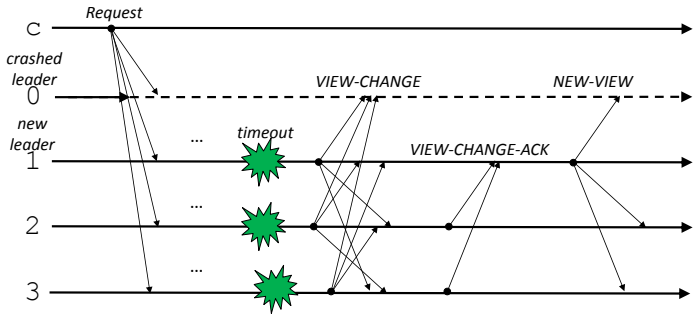

Figure 4.7. PBFT view change protocol.

Recovery. The PBFT recovery protocol comprises several steps, some of them related with the execution of proactive recovery for bounding the effect of malicious faults in the system (Castro and Liskov 2002). In this section we present a simple adaptation of this protocol for the case when a replica crashes and recovers and then tries to integrate itself back in the system.

The recovery is done in two steps. The recovering replica first tries to discover which is the most recent sequence number i committed in the system. This can be done by listening to the messages exchanged in the system or actively probing the other replicas. When this number is discovered, the replica disseminates a FETCH request asking for the log of processed requests up to i. When a replica receives this message, it replies with the requested state. The recovery replica installs the new state and resumes operation only when it receives $f + 1$ matching states from other replicas. For a more in depth discussion of this kind of state transfer protocol, please refer to (Bessani et al. 2013).

4.3.3. MinBFT and the use of Trusted Components

MinBFT (Veronese et al. 2013) is a BFT SMR protocol that requires the same number of replicas and communication steps as VR/Paxos (Section 4.3.1). This is achieved through the use of small trusted components in a replica, under a hybrid fault model (Verissimo et al. 2003). By employing fewer replicas, MinBFT reduces the costs in hardware, software and administration. Reducing the communication steps makes the system perform better, especially in wide-area networks, where latencies are significantly higher.

Fault model. MinBFT requires $n = 2f + 1$ replicas, in which up to f can be subject to Byzantine faults. However, each replica is equipped with a small trusted component that cannot be compromised, even in faulty replicas, being thus subject only to crash faults. This component, called USIG (Unique Signed Identifier Generator), implements a trusted counter that can be used to ensure that malicious replicas do not send conflicting messages. The USIG provides the following interface to the replica (Veronese et al. 2013):

- `createUI`(m) – returns a *USIG certificate* that contains a *unique identifier UI* and certifies that this *UI* was created by this tamperproof component for message m. The unique identifier is essentially a reading of the USIG monotonic counter, which is incremented whenever `createUI` is called.
- `verifyUI`(PK, UI, m) – verifies if the unique identifier *UI* is *valid* for message m, i.e., if the USIG certificate matches the message and the rest of the data in *UI*.

In the MinBFT paper this component is implemented in two ways: (1) using an isolated virtual machine in each physical replica and (2) using a Trusted Platform Module (TPM) chip. (1) provides better performance but requires trust in the VM software stack, while (2) presents a sufferable performance, but is based on a secure chip that is commonly available in modern machines. Solution (2) is considered more secure because the keys and the security functionality are stored in a dedicated co-processor, separated from the main system, with hardware-enforced isolation. More recently, a FPGA-based implementation of a trusted component similar to the USIG was shown to achieve excellent performance (Kapitza et al. 2012) without sacrificing strong security.

Normal Case. MinBFT has only two communication steps, not three like PBFT (Section 4.3.2), with a message pattern (illustrated in Figure 4.8) very similar to VR with fast execution (see Figure 4.4). The following steps are executed in the normal case of the protocol:

1. The client sends a signed request to all replicas;
2. The leader assigns a sequence number (execution order number) to the request and sends it to all replicas in a PREPARE message. The assigned sequence number is the counter value in an *UI* returned by the USIG service (through the `createUI` operation), thus the leader can not repeat or assign arbitrarily higher sequence number to a request;

3. Upon the reception of a PREPARE message from the leader, other replicas verify the associated UI (using the `verifyUI` operation) and sends a COMMIT message to all other replicas. In order to constraint the actions of malicious replicas, these messages also carry a unique identifier returned by the USIG service to ensure malicious replicas are not allowed to send conflicting COMMIT messages;

4. When a replica receives $f+1$ COMMIT messages for a request, it executes the request and sends a reply to the client;

5. The client waits for $f+1$ matching replies for the request to completes the operation.

Notice that during the normal execution of the protocol, each correct replica needs to create exactly UI, but only the UI created by the leader defines the order of the request (Veronese et al. 2013).

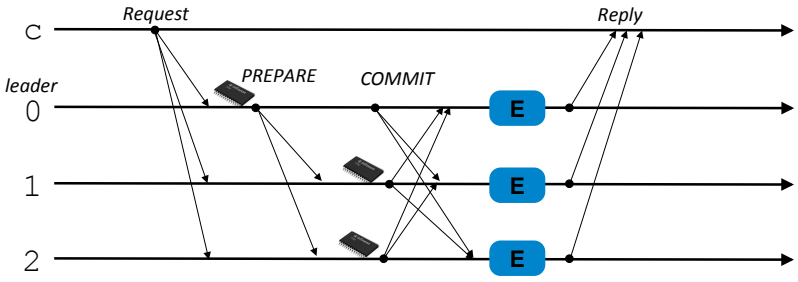

Figure 4.8. MinBFT normal case execution. The "chip" symbol represents the creation of an UI by the USIG.

View Change. When $f+1$ replicas suspect that the leader is faulty, a view change protocol is executed in order to select a new leader. As with previous protocols, view changes are triggered by a timeout: when a non-leader replica receives a client request, it starts a timer. When the request is executed, the timer is stopped. If the timer expires before this, a backup replica suspects that the leader is faulty and starts a view change, which works in the following way (illustrated in Figure 4.9):

1. When the timer for a request expires in some replica, it asks for a view change by sending a REQ-VIEW-CHANGE message to all other replicas;

2. When a replica receives $f+1$ of such messages, it moves to the next view and sends a VIEW-CHANGE message to all other replicas, informing the last request executed in the previous view. To prevent faulty replicas from sending different VIEW-CHANGE messages to different replicas, these messages also carries an unique identifier returned by the USIG service;

3. When the new leader receives $f+1$ VIEW-CHANGE messages, it defines the initial state for the next view from the information received and sends a NEW-VIEW message to all other replicas (this message carry a certificate generated from the VIEW-CHANGE messages received and an unique identifier returned by the USIG service);

4. When a replica receives a NEW-VIEW message with a valid certificate it starts the next view.

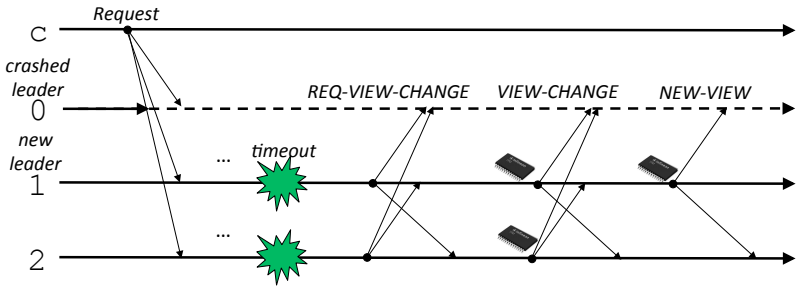

Figure 4.9. MinBFT view change protocol.

Recovery. The original MinBFT protocol does not assume replicas can recover. In principle, it is possible to apply the same ideas as in the PBFT recovery protocol, however, additional care must be taken to ensure that the recovered replica discovers the next expected UI counter value for each other replica before resuming the normal case operation.

4.3.4. Optimizations and Extensions

In the following we briefly describe two general optimizations and an additional protocol that can be integrated to any of the baseline protocols presented in this section.

Batching. In order to amortize the cost of executing the ordering protocol for each operation requested by clients, the protocols can be used to order batches of requests that, once ordered, are delivered in a deterministic order in each replica. This optimization can be applied in all three protocols and may lead to significant performance improvements when there are many clients and requests are small.

Digest replies. This optimization makes sense only for BFT protocols, in which clients need to receive multiple matching replies for tolerating faulty replicas. The key idea here is, instead of all replicas sending the reply for a request, the client can choose just one of the replicas to send the reply while the others send only a digest of the reply for voting the result. If the received reply is wrong, the client can ask for a (full) reply from other replicas (Castro and Liskov 1999).

Reconfiguration. Reconfiguration is the process of changing the set of replicas that comprise the SMR system. This kind of protocol is important to ensure failed replicas can be replaced at runtime. Through reconfigurations, the system moves from some configuration (a set of replicas) to another updated configuration. Although the original paper on the Paxos protocol already defined a method for reconfiguring a replicated state machine (Lamport 1998), only recently this kind of protocol gained more attention (Lee and Thekkath 1996; Lorch et al. 2006; Lamport et al. 2010; Liskov and Cowling 2012).

The simplest way to implement reconfigurations is by using the replicated state machine itself (running an old configuration) to specify the new configuration, making all replicas agree about what to change (Lamport 1998; Lamport et al. 2010). Following this approach, when a reconfiguration is required, a special reconfiguration request is issued to the RSM[10]. This request goes through the ordering protocol as a normal client request, in a way that every replica will execute it in the same order. A replica that executes the reconfiguration request does not change the application state, instead it goes to the next configuration. Consequently, the reconfigurable system can be seem as a sequence of replicated state machines ($RSM_0, RSM_1, ...RSM_n$), where both the configuration and the initial state of RSM_{i+1} is defined by RSM_i.

4.4. Extensions and Recent Work

This section presents some extensions proposed to the SMR baseline protocols. Beyond trying to optimize the latency of replication allowing its execution in a wide-area, these extensions aim to improve performance, modularity and robustness of SMR protocols.

4.4.1. Improving Performance

The many flavors of Paxos. There are many extensions and optimizations proposed to the Paxos/VR algorithm. Here we cite two examples. Fast Paxos (Lamport 2006) is a version of Paxos in which commands can be committed in two communication steps if there are no concurrent updates being executed. Ring Paxos (Marandi et al. 2010) is a recent protocol that exploits the characteristics of IP multicast networks for implementing efficient replica coordination (i.e., total order multicast) in this environment.

Beyond PBFT. After the publication of PBFT, several other works devised new protocols (Abd-El-Malek et al. 2005; Cowling et al. 2006; Kotla et al. 2009) that present improved performance under certain conditions. Although these protocols were designed for tolerating Byzantine faults, the ideas employed here can be easily adapted for algorithms tolerating only crashes.

The first of these algorithms is Q/U (Abd-El-Malek et al. 2005), a pure quorum-based protocol which executes updates in one roundtrip under certain conditions and achieves better scalability with big replica groups. Since this cannot be done ensuring wait-freedom, the approach sacrifices liveness: an operation is guaranteed to terminate only if there is no other operation executing concurrently on the same object. The main benefit of Q/U is its fault-scalability: it attains high throughput even when the number of faults tolerated is high. Moreover, since only basic quorum protocols are employed, the Q/U protocol has linear message complexity and small expected latency (two communication steps) when there is no contention or failures. The drawback of this approach is that it requires $n \geq 5f + 1$ replicas and provides only Obstruction-freedom (Herlihy et al. 2003), a weaker termination condition than wait freedom (Herlihy 1991).

HQ-Replication was the first known protocol to integrate pure read/write quo-

[10]Only a subset of clients might be authorized to ask for reconfigurations, separating this interface from the one used for normal client requests.

rum protocols with total-order broadcast for implementing efficient state machine replication (Cowling et al. 2006). The basic idea is to use only quorum protocols when there is no contention in an object access and agreement protocols to resolve situations of contention. In HQ, a process reads and updates, respectively, in two (four when write-back is needed) and four communication steps in contention-free executions. When contention is detected (in reads too, due to the write-back), the system uses the PBFT protocol to order contending requests. This contentious resolution protocol adds great latency to the protocol, reaching more than ten communication steps even in synchronous and failure-free executions.

Kotla et al proposed Zyzzyva, a *speculative* BFT state machine replication that can order requests in only three communication steps (Kotla et al. 2009). This protocol uses the result from Martin and Alvisi (Martin and Alvisi 2006) in the following way: after receiving a proposal from the leader, the replicas speculatively execute the client request and send the response to the client, which act as a learner. The client receives responses from the servers and knows if the result was correct, but a server does not known if its update is in accordance with other servers until a explicit commit is received. Such commit is exchanged by the replicas periodically. If the received replies match, the operation is concluded, otherwise a recovery procedure (that may lead to a view change) is triggered by the client.

The server state is called *speculative* since they do not know if the other servers are processing the same operation in the same order. Consequently, they need to store the previous state to be rolled back if some inconsistencies are found during the speculative execution. The resulting protocol executes very efficiently in synchronous environments when there are no faults, requiring only three communication steps with linear message complexity. However, there are two fundamental drawbacks in the approach. First, speculation may be difficult to implement in some systems in which executed commands cannot be undone. Second, a client needs to wait for $n = 3f + 1$ speculative replies, instead of the $f + 1$ or $2f + 1$ replies waited in PBFT. This can be a problem in wide area networks.

4.4.2. Modularity

Protocol Composition. Guerraoui et al. (Guerraoui et al. 2010) proposed ABSTRACT, a composable subprotocol abstraction that allows the integration of optimizations, such as the ones introduced in HQ, Q/U and Zyzzyva, in a modular way in a conservative protocol such as PBFT. More precisely, ABSTRACT allows the composition of *abortable subprotocols* that execute successfully only if some optimistic properties are satisfied in the execution, aborting otherwise. For example, a Zyzzyva-like subprotocol would succeed only if there are no faults in the system and if the execution is synchronous. If some of these conditions are not satisfied the protocol aborts and gives to the client a proof that something went wrong. This proof can potentially be used to initiate another subprotocol that deals with faults or asynchronous executions.

Using this abstraction, a novel BFT SMR protocol called Aleph was proposed. Aleph is composed of three subprotocols, as described in Figure 4.10. Clients start using a *Quorum* subprotocol that is latency-optimal and works only if there is no contention. When contention is detected, Quorum aborts and a new subprotocol called *Chain* enters in

operation. Chain is ring-like protocol that is throughput-optimal as long as the execution is synchronous and there are no faults. If one of these conditions is not satisfied, Chain aborts and the non-abortable *Backup* subprotocol is used. Backup implements PBFT with some minor additions for compatibility with other protocols.

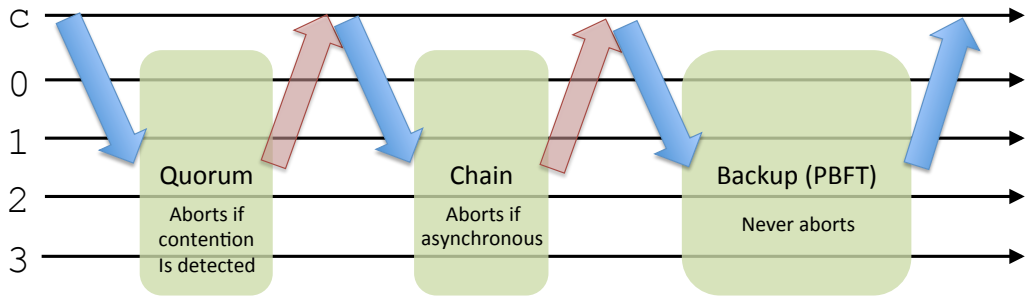

Figure 4.10. Aleph in action.

The notion of integrating subprotocols devised in this work is quite powerful and, although proposed in the context of general Byzantine fault tolerance, can be used in other fault models as well. For example, Kapitza et al (Kapitza et al. 2012) introduced the CheapBFT protocol for efficiently tolerating Byzantine faults in systems in which replicas are equipped with a trusted component. This protocol requires only $f+1$ active replicas in synchronous and faulty-free executions plus f passive replicas. However, this protocol may abort and the system uses the full set of $2f+1$ replicas to run the MinBFT protocol, in the same way as Aleph uses PBFT as Backup.

A similar approach to Aleph or CheapBFT could also be applied to crash fault-tolerant protocols, using Paxos/VR as a non-abortable Backup protocol.

Separating Agreement from Execution. In a classical RSM, a client application invokes an operation that is sent to one or more replicas that (1) reach agreement about the ordering of the operation and (2) execute the command, sending the reply to the client.

The last row of Table 4.1 shows that in a partially synchronous distributed systems with Byzantine faults, $3f+1$ replicas are required to implement replica coordination, but only $2f+1$ replicas are needed to execute the requests[11]. Yin et al (Yin et al. 2003) exploited this idea and proposed the separation of the system in two different layers of servers: agreement and execution. The agreement layer, with $3f+1$ servers, is responsible for ordering requests and send them to the execution layer in total order, with $2f+1$ servers, that execute the request and send back a reply to the agreement layer that relay it to the clients that invoked the operations. This architecture is illustrated in Figure 4.11.

This methodology has two main benefits. First, it reduces the amount of resources required for implementing application servers (disk, memory and CPU), since now the service code needs to be deployed in $2f+1$ servers. Second, it allows the use of the same ordering layer as an ordering service (Kapritsos and Junqueira 2010), which can be used by many replicated applications.

[11] A similar observation can be made about crash faults on the first row of the table, and thus similar optimizations can be implemented in this fault model.

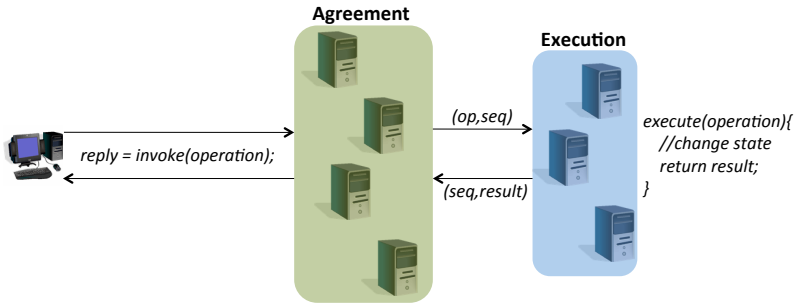

Figure 4.11. The separation of agreement and execution in different server layers.

Two more recent works extended the separation in different directions. Wood et al proposed a system called ZZ which exploits hypervisor technology for using only $f+1$ execution replicas synchronous and fault-free executions (Wood et al. 2011). When such conditions do not hold, f extra replicas are started as virtual machines and used just like in (Yin et al. 2003). Notice this approach works very well in modern cloud settings (where VMs are the norm) under the assumption that the cloud manager is trusted, i.e., under the assumption that it is possible to start non-compromised correct VMs when needed.

Another work that refines the separation of layers in RSMs was done within the context of the UpRight replication library (Clement et al. 2009a). This work proposes the division of agreement layer in two layers: request quorum and order. The request quorum layer (or stage, as it is called in the paper), which requires $2f+1$ servers and is responsible to (1) validate client requests and (2) separate the data path from the control path. The benefit of (1) is to avoid corner cases related with the presence of Byzantine clients while the big advantage of (2) is to avoid sending (big) requests to the ordering layer. (2) is of particular importance since agreement protocols are quite sensitive to request size. For instance, BFT-SMaRt (which will be described in Section 4.6) can order almost 79000 100B-requests/sec, but less than 17000 1kB-requests/sec (Bessani et al. 2014).

Durability. One feature commonly required in practical SMR-based systems is the Durability of replicated services. Durability is defined as the capability of a SMR system to survive the crash or shutdown of all its replicas, without losing any operation acknowledged to the clients. Its relevance is justified by the need for planned maintenance actions and also by the many examples of significant failures that occur in data centers making many servers to crash simultaneously (e.g., (Ford et al. 2010; Miller 2008)).

The integration of durability techniques such as logging, checkpointing, and state transfer with the SMR approach can drastically decrease the performance of a service. In particular, synchronous logging can make the system throughput as low as the number of appends that can be performed on the disk per second, typically just a few hundreds. Additionally, checkpointing requires stopping the service during this operation, unless non-trivial optimizations are used at the application layer, such as copy-on-write (Clement et al. 2009a; Hunt et al. 2010). Finally, recovering faulty replicas involves running a state transfer protocol, which can degrade the performance of the system as correct replicas need to transmit their state.

A recent work proposed the introduction of a layer between the replication protocol and the service code to implement the durability techniques (Bessani et al. 2013). This durability layer implements efficient techniques for logging, checkpoints and state transfer. These techniques allow the implementation of synchronous disk logging with the same throughput of a pure in-memory system (for 4kB-requests) with minimal performance perturbations caused by checkpoints and state transfers.

Even more importantly, the durability techniques are confined inside this new layer, making the service code completely oblivious to the durability implementation as long as the service state fits in the replica memory. The key takeaway from this is that it is possible to integrate durability to a SMR-based system without application code with proper support from the system.

4.4.3. Improving Robustness of BFT State Machine Replication

Dealing with Malicious Byzantine faults. In order to tolerate Byzantine faults caused by a malicious and intelligent adversary controlling the faulty machines, at least three extensions can be made to a BFT SMR protocol (Bessani 2011). First, the replicas need to be deployed in a diverse environment to ensure that a single vulnerability does not lead to more than one replica being compromised (as discussed in Section 4.2.3.1). Second, the replica coordination protocol need to be resilient to certain known performance degradation attacks (see bellow). Finally, the system needs to be augmented with means to recover faulty replicas to enable the system to tolerate an unbounded number of faults during its lifetime.

Proactive recovery (Castro and Liskov 2002) aims to periodically and proactively rejuvenate replicas even if they are correct to clean potential undetected intrusions. The main advantage of these systems is that the adversary can stay in control of some replicas only during a window of vulnerability, even if its perpetrating a stealth intrusion and could not be detected.

Recent works showed that many proactive recovery systems can be attacked and delayed, giving time for an intelligent adversary to compromise more than f machines and takeover the system (Sousa et al. 2007). To deal with these vulnerabilities it is necessary to increment the replicated system with a synchronous subsystem capable of triggering timely recoveries without interference of attackers (Sousa et al. 2010).

Robust replica coordination. Amir et al (Amir et al. 2008) showed that PBFT and other leader-based BFT replication systems are vulnerable to two attacks that can seriously degrade the performance of the replicated system, even if less than f replicas are compromised. The first attack, called *Pre-Prepare delay*, exploits the fact that non-leader replicas only start the view change protocol if a timeout is triggered. This timeout is usually defined in a conservative way, with a value of tens of seconds, while a request ordering takes few milliseconds (or even less). This means that a malicious primary can consistently delay the ordering of messages more than an order of magnitude without being detected. This attack can be made even more devastating when coordinated with a second attack, dubbed *timeout manipulation*, which exploits the fact that malicious clients can force a leader change in PBFT. Furthermore, after each view change the timeout value

is doubled. This means that a malicious client can increase a timeout value as much as it wishes and only stops when a faulty replica is the leader, forcing the system to have a malicious leader.

As far as we know, these attacks can be mitigated in three possible ways. Prime is the protocol proposed in the same paper the attacks were identified (Amir et al. 2008). This replication algorithm introduces an additional pre-order phase with three communication steps before the request ordering (which are based on PBFT) that, together with the constant monitoring of the performance of the primary, make the system able to detect several performance attacks and change the leader when it degrades the performance of the system.

The second solution is Aardvark (Clement et al. 2009b), a BFT library in which a set of engineering principles are applied to PBFT to make it more resilient against several kinds of attacks from clients and servers. One of the attacks addressed by Aardvark is the pre-prepare delay injected by a malicious primary. This attack is mitigated through constant monitoring of the throughput sustained during a view plus the periodic change of primary through the execution of PBFT's view change operation. Furthermore, clients cannot force a leader change (and an increase in timeout values) since replicas only accept requests signed by clients. This means that a request is valid or invalid in all correct replicas, and thus no correct replica will suspect a correct leader that discarded an invalid request.

The third and last solution is based in distributing the burden of leadership among the replicas in such way that a malicious replica can delay requests in only $\frac{1}{n}$ of the agreement executions. The Spinning protocol (Veronese et al. 2009) exploits this idea through the use of a rotating coordinator. To avoid malicious replicas periodically imposing outrageous delays to the protocol, the protocol uses a monitoring strategy and a blacklisting mechanism.

Overall, independently on the strategy adopted for avoiding pre-prepare delays, it is advisable to request clients to sign requests if malicious behavior is anticipated.

4.4.4. Wide-area Replication

There are some recent works that try to optimize the latency of the replication in geo-replicated state machines (e.g., (Mao et al. 2008; Moraru et al. 2013; Veronese et al. 2010)). One of the most important differences between these protocols and the protocols of Section 4.3, is that they do not use a single stable leader, that is changed only in case of problems, and instead, allow every replica to act as a leader for the clients that are close to them. This brings two advantages. First, the load of being a leader is distributed among the replicas and their sites, avoiding having the leader as a bottleneck in the system. Second, if the client submits its commands (and get the reply) to a replica that is close to itself, it avoids at least one wide-area communication step.

Mencius (Mao et al. 2008) extends Paxos through the division of the sequence number space between the set of replicas, allowing each replica to propose commands to be executed in its slots. In this way, in a system with 3 replicas, replica 1 can propose commands for sequences 1, 4, 7, etc. while replica 2 can use slots 2, 5 and so on. To deal

with the cases in which a replica does not have anything to propose in its next slot, Mencius introduces a new Skip message, which is disseminated by a replica in this situation. EBAWA (Veronese et al. 2010) extends MinBFT in a similar way that Mencius extends Paxos for performing better on WANs. More recently, Moraru et al proposed Egalitarian Paxos (EPaxos) (Moraru et al. 2013), a crash fault-tolerant leaderless state machine replication algorithm in which only dependent commands are executed in total order.

4.5. SMR Applications

In this section we discuss some recent papers that describe the use of different SMR algorithms for implementing dependable services. Some of these papers report on real systems that are used in production, while others are based on proposals for innovative systems to be deployed in the future.

4.5.1. Dependable Storage Systems

The most direct application of RSMs is in the implementation of dependable storage systems. Such systems can be as simple as main-memory key-value stores or as complex as full-fledge file systems and transactional relational databases.

4.5.1.1. Key-Value Stores

Several basic data store systems, supporting a simple key-value store (KV-Store) interface were described in recent literature (Bessani et al. 2013; Bolosky et al. 2011; Rao et al. 2011; Wang et al. 2012). There are many variants and capabilities associated with KV-Stores, which are usually related with the NoSQL model. However, the simplest variant assumes a simple data model in which a table is used to map keys to binary values. This table supports operations such as $put(K,V)$ (stores value V associated with key K), $get(K)$ (gets the value associated with K), $remove(K)$ (remove K and its associated value from the store) and $list()$ (lists all stored keys). In the following we discuss the implementation of these operations under different assumptions.

Given a SMR protocol and programming library, a trivial implementation for a dependable KV-Store can be achieved using a main-memory deterministic data structure in each replica. Assuming all replicas start the system with the same values stored in such data structure, the updates can be applied in total order (as provided by the SMR protocol) to ensure all replicas have the same state. In Section 4.6 we show how to implement a service like this in Java using the BFT-SMART replication library and a `java.util.TreeMap` data structure.

In terms of dependability, this kind of system works under two assumptions:

- **DS1:** the stored data fits on a replica main memory;
- **DS2:** at any given time, there is at most f faulty replicas.

These two assumptions are a direct consequence of the fact that our implementation is completely based on main memory, with no use of stable storage.

A more evolved implementation would use stable memory for ensuring data durability, i.e., to guarantee that the stored data is recovered even if more than f faults happens in the system. In practice this property is quite important since sometimes the system can be shut down for maintenance reasons or the replicas can suffer correlated failures. Consequently, restarts and full-system recoveries are needed (Bessani et al. 2013).

In a durable system, every operation that updates the state (e.g., *put* and *remove* for a KV-Store) of the system is logged in stable memory[12] before a client is notified about its result. To limit the size of this log, periodically the system takes a snapshot of its state and store it in stable memory, deleting the log. The period of checkpoint is usually based on the log size or the number of operations processed, not the time elapsed. In this sense, the replica stable state comprises the log and snapshot, which can be recovered after a crash.

Notice that if these techniques are employed, the data still needs to fit in memory (property DS1, above), since the stable storage is used only for recovery. However the system now can offer better guarantees than what was provided by DS2:

- **DS2+:** the system is live (i.e., satisfy liveness) as long as there is at most f faulty replicas.

Notice DS2+ is now a direct consequence of the requirements for executing the agreement protocol of the system, i.e., if there are more than f faulty replicas, it is impossible to order requests and thus the RSM cannot execute operations. Furthermore, if Byzantine faults are considered, this guarantee is ensured only if the faulty replicas suffer a crash, and not a state corruption. A BFT system that implements this kind of design is the SCKV-Store (Bessani et al. 2013).

If we want to remove the limitation of having the whole state in main memory, and thus improve the system for storing large amounts of data, one needs to employ secondary storage data structures, as used in database manage systems (Garcia-Molina et al. 2009). One possible implementation of such system is the one used in the Paxos-based Gaios data store (Bolosky et al. 2011). This system still uses the log as described before, and uses the main-memory data structure as a cache in which operations are executed. Later on, in background, the updates are pushed to the disk in batches. Notice that updating the storage in background is not a problem since the durability is ensured by the logging.

An interesting innovation of Gaios is how it uses the VR/Paxos leader for distributing read requests among replicas. This technique is important specially in disk-based systems, where the latency of disk readings can be an order of magnitude bigger than running the agreement protocol (e.g., 5 vs. 0.5 milliseconds). The basic idea is to use the disk of a single replica for serving a read-only request. The protocol requires a client to send the read request to the leader, that choses one of the replicas for answering and forwards the request to it. The choice of the read-serving replica is based on a load balancing policy (e.g., round robin) and also in trying to avoid assigning requests to repli-

[12] Such IO operation must be done in a synchronous way to ensure the write will be on stable storage when the system call returns.

cas flushing writes to disk. Importantly, this algorithm ensures no read returns stale data, since even read operations are ordered by the leader (Bolosky et al. 2011).

There are also complex, scalable distributed data stores that use state machine replication protocols for ensuring strong consistency for the stored data (Baker et al. 2011; Calder et al. 2011; Corbett et al. 2012). Although the details about the design of these systems are out of the scope of this chapter, it is interesting to understand how SMR protocols are used in their core.

The key idea that makes these systems scalable is the partition of the key space in different RSMs (i.e., Paxos group (Corbett et al. 2012)) and deploy and operate these sub-systems as independently as possible. An important issue that needs to be dealt with in this design is how to implement operations that affect more than one key space. The standard approach, employed for instance in Google's Spanner (Corbett et al. 2012), is to execute a transaction between the several Paxos groups involved in the operation. Naturally, running this transaction requires coordination between the different groups. In the case of Spanner, the leaders of each Paxos group involved in the transaction participate in a two-phase commit protocol execution to decide the outcome of the transaction.

4.5.1.2. File Systems

The same kind of techniques used for implementing KV-Stores can also be used for implementing SMR-based file system, such as Harp (Liskov et al. 1991), which is based on viewstamped replication, and BFS (Castro and Liskov 2002), based on the PBFT.

Ignoring some implementation details, conceptually, implementing a file system service requires implementing the metadata store and the data store. The metadata store is where file names together with access times, file size and access control information are stored together with the ids of the blocks corresponding to the file content. The file content is stored in a data store, having the block' id as a key and its contents as the associated value. The data blocks usually have fixed size (e.g., 4kB (Liskov et al. 1991; Castro and Liskov 2002)), that can only be read or write in indivisible blocks. The RSM implements the metadata and data stores together, as a service, to offer a NFS-like interface for its clients, which provides a transparent remote file access.

A more scalable approach for serving files is the one employed in systems like FARSITE (Adya et. al 2002) and UR-HDFS (Clement et al. 2009a). In a nutshell, the key idea of these scalable systems is to separate the metadata store from the data store, as done in parallel file systems (Gibson et al. 1998). This separation allows the use of RSM for implementing the metadata service, which stores only file names and associated metadata, and thus can be kept even in main memory as discussed for KV-Stores. The file contents, which can comprise a huge amount of storage space, are stored in a potentially large amount of machines (or even network attached disks) that are accessed directly upon reading and writing.

Figure 4.12 shows the architecture of these systems. In the figure, a client reading a file (left) first goes to the metadata service and read the location of the block(s) of the file. This location includes the storage node, the id of the block and the hash of the block

contents. With this information the client can fetch the contents of the file from one (or more) storage nodes in which the data is stored. After reading the contents of a block, the client verifies its integrity with the hash obtained from the metadata store. To write a file, the client first obtains the locations of the free blocks in the storage nodes from the metadata store and then writes the file contents in these locations. After completing the write, the data nodes send a hash of the written data to the metadata store. To avoid concurrency problems, when the metadata service allocates these blocks to the client, they are locked until the data node informs the hash of the write to the system or a timer expires.

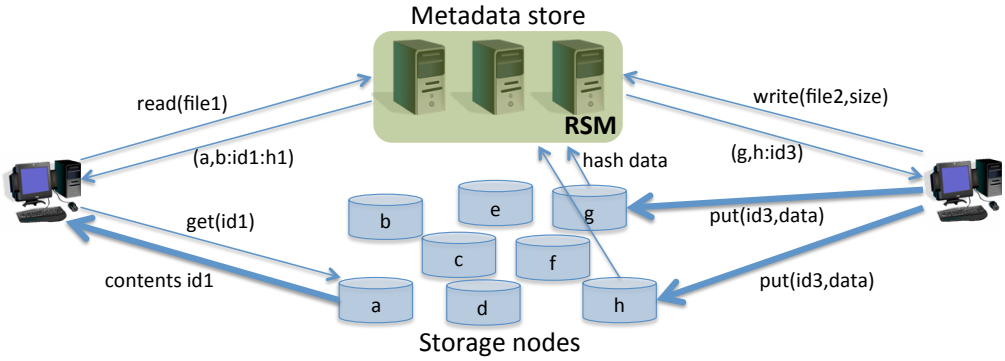

Figure 4.12. Scalable file system design based on state machine replication.

4.5.1.3. Transactional Storage

One limitation of the state machine replication model is that it ensures strong consistency (i.e., linearizability (Herlihy and Wing 1990)) only considering single operations. However, popular storage systems provide the capacity of running atomic transactions (Garcia-Molina et al. 2009). Transactions are sequences of operations that are either executed in totality or not executed at all. Another important property of transactions is the isolation: two transactions executed concurrently generate the same result as if they are executed serially, one after another. A problem in implementing such property with SMR is that it requires locking the state touched by a transaction, and it may cause the system to block. More specifically, a deadlock may be caused by two concurrent transactions accessing the the same data objects in the RSM in different order.

There are some solutions for this problem, but in general, the best compromise is achieved when one restrains to provide strict serializability and offer instead a property called snapshot isolation (Berenson et al. 1995). According to the definition of snapshot isolation, reads on multiple versions of a database are admitted and two concurrent transactions will commit if there are no write-write conflicts. There are algorithm for SMR-based database replication satisfying snapshot isolation for tolerating both crashes (e.g., (Elnikety et al. 2005)) and Byzantine failures (Garcia et al. 2011). In the following we describe the Byzantium algorithm (Garcia et al. 2011), which tolerates Byzantine faults using normal off-the-shelf databases at each replica.

The key idea of the algorithm is to invoke SMR ordered operation only at the

borders of a transaction and accessing a single replica during the operations within the transaction. The algorithm works as follow:

1. When a transaction starts, the client issues a SMR invocation for the replicas to mark a snapshot of the database. The transaction operations will be executed in this snapshot;

2. During the transaction, the client executes the operation only in one of the replicas, the master (but not necessarily the leader of the SMR protocol);

3. For committing the transaction, the client issues an SMR operation to the replicas passing the operations executed and a hash of the obtained results. The replicas then verify if the operations were correctly executed by the master and try to commit the transaction in the database, considering the snapshot marked at the beginning of the transaction.

If for some reason the transaction aborts due to malicious behavior of the master, the transaction aborts and the client asks the replicas to elect a new master (Garcia et al. 2011).

It is worth to remark that a similar (albeit simpler) algorithm exists for tolerating only crashes (Elnikety et al. 2005). Furthermore, the approach sketched here can be applied for supporting transactions in general SMR-based services, not only relational databases.

4.5.2. Coordination Services

Coordination services (also called lock services or configuration managers) are small databases that offer support for the coordination of other applications on an infrastructure. Such services are used for storing configuration data and to implement resource locking, leader election, message ordering, among other synchronization tasks. The big advantage of using a coordination service is that it relieves distributed application developed to implement complex and error-prone synchronization algorithms, providing instead a central point of coordination in the system.

Naturally, such service will be a critical component of the distributed system, and thus it should be designed in a dependable way. SMR-based techniques are often employed for that (Burrows 2006; Hunt et al. 2010; Bessani et al. 2008).

In a nutshell, a coordination service is a durable main-memory storage system that offers operations with synchronization power (Herlihy and Wing 1990). In this way, although the data stored in the coordination service must still fit in a replica memory, durability techniques ensure that such data will not be lost in case there are more than f faulty replicas. In the remaining of this section we describe three representative coordination services.

Chubby is a locking service by Google (Burrows 2006) that implements a hierarchical name space with locking primitives. Chubby exports a file system interface in which users can create small files that can be locked for a specific amount of time. This

abstraction is implemented using the Paxos protocol, with leader replicas having the additional responsibility of notifying clients about events of interest in the system (see below).

Another interesting aspect of chubby is the client-side cache it offers. This cache absorbs a large amount of read requests and relieves the coordination service replicas from a significant work, contributing to the system scalability. However, as a consequence, a cache invalidation protocol is used, and the leader replica is responsible for notifying clients about cached data updates and lock expirations (Burrows 2006). This kind of design breaks the modularity of a pure SMR-based system, in which the ordering protocol is completely independent of the service being replicated.

Apache ZooKeeper is a widely used open source coordination service that provides a hierarchical name space and strong coordination primitives (Hunt et al. 2010). However, differently from Chubby, ZooKeeper provides no blocking primitives such as locks, and instead only wait-free objects are provided in its data model.

More specifically, coordination is implemented in ZooKeeper through the use of znodes, which have a name and value associated. Znodes can be regular or ephemeral. An ephemeral znode disappears from the ZooKeeper namespace when the client that creates it disconnects (e.g., due to a crashes), which is quite useful for detecting failures. Regular nodes can have other znodes as children. A child node can be created using the sequential flag, which ensures that every node created under the same parent will have monotonically increasing sequence number appended to its name. This basic feature allows Zookeeper data model to have synchronization power sufficient for solving consensus and related tasks, such as leader election and message ordering (Hunt et al. 2010).

Similarly to Chubby, in ZooKeeper the master has additional responsibilities besides the ordering of requests. Consequently, the durability mechanisms, data store implementation and message ordering are highly integrated, with little or no separation between them. Due to this, ZooKeeper requires a Paxos-like replication protocol called Zab (Junqueira et al. 2011), which offers additional guarantees that are important for systems in which the primary is specially important.

Both Chubby and ZooKeeper tolerate only crash faults. Although there were some prototypes for adapting ZooKeeper for tolerating Byzantine faults (e.g., UR-ZK (Clement et al. 2009a)), the only coordination service genuinely tolerant to Byzantine faults is DepSpace (Bessani et al. 2008). This system provides a tuple space abstraction (Gelernter 1985) in which variable size data structures called tuples are inserted, read and removed. In particular, DepSpace provides additional operations to allow a tuple space to have sufficient synchronization power for solving consensus (Bessani et al. 2009).

Contrary to Chubby and ZooKeeper, the implementation of DepSpace is modular with respect of the SMR protocols: the coordination service code is implemented on top of the replication algorithm. Notice that although DepSpace originally has no support for durability, recently it was updated for integrating efficient durability techniques (Bessani et al. 2013).

4.5.3. Network Services

Recently there has been an effort to separate the network control plane from the data plane for improving the manageability, programmability and extensibility of networks. Most of these efforts are being done in the context of Software-Defined Networks (SDNs). In this paradigm, a programmable control plane is implemented through one or more controllers attached to the network switches that act as simple forwarding devices. These controllers host network applications capable of deciding, for each flow, how the network should behave.

One key requirement of these systems is that the network controller should be neither a bottleneck nor a single point of failure. This means that the design of such systems should be scalable and fault-tolerant. One appealing solution is to use a distributed controller integrated with a fault-tolerant network information base (NIB) (Koponen et al. 2010; Botelho et al. 2013). Since the NIB is not expected to store a lot of information, it can be implemented just like a KV-Store, presented in Section 4.5.1, with controllers acting as clients. This kind of design was recently shown to be able to deal with representative SDN workloads for small to medium networks (Botelho et al. 2013).

Other distributed network services have been using SMR-like techniques for implementing middlebox-related functionalities (e.g., NAT, VPN, load balancing) without using middleboxes. For example, the ETTM system (Dixon et al. 2011) implements such functionalities in a distributed way using a Paxos-based replication algorithm for storing the service-related state in a consistent and fault-tolerant way. In a production environment, the Ananta distributed load balancer (Patel et al. 2013), which serves thousands of flows per day in the Windows Azure cloud, uses Paxos for maintaining high-availability in its manager component, which keeps the configuration of individual managed load balancers (called MUXes).

4.5.4. Other Services

There are many other applications of state machine replication in practical systems. For example, BFT state machine replication has been used to implement survivable DNS systems (Cachin and Samar 2004) and even a SCADA system (Kirsch et al. 2013). When only crash faults are considered, the Paxos/VR algorithm is becoming a de-facto standard with application in basically every critical system deployed these days.

4.6. Implementing SMR: The BFT-SMaRt Library

The last part of this chapter describes some details about the internals of the BFT-SMaRt SMR replication library, its performance and how it can be used for implementing dependable services. The objective here is to show how some of the techniques described through the chapter are implemented in practice. In this way, we just highlight selected aspects of the system and refer the interested readers to the papers describing it for details (Sousa and Bessani 2012; Bessani et al. 2013; Bessani et al. 2014).

BFT-SMaRt is an open-source Java-based library implementing robust BFT state machine replication. Some of the key features of this library that distinguishes it from similar systems (e.g., PBFT (Castro and Liskov 2002) and UpRightt (Clement et al. 2009a))

are its reliability, modularity, multicore-awareness, reconfiguration support and a flexible API. When compared to other SMR libraries, BFT-SMART achieves better performance and is able to withstand a number of real-world faults that previous implementations cannot.

4.6.1. BFT-SMART Design

The design principles that guided the development of BFT-SMART are: *(1) tunable fault model* – it is possible to configure the system to tolerate Byzantine failures or only crash failures; *(2) simplicity* – the emphasis on protocol correctness led BFT-SMART to avoid optimizations that could bring extra complexity both in terms of deployment, coding, or even new corner cases; *(3) modularity* – BFT-SMART implements a modular SMR protocol that uses a well-defined consensus primitive in its core, besides the existence of modules for reliable point-to-point communication, client requests ordering and consensus, BFT-SMART also implements state transfer and reconfiguration modules, which are completely separated from the agreement protocol; *(4) simple and extensible API* – BFT-SMART encapsulates all the complexity of BFT SMR inside a simple API that could be extended in order to implement a more complex application; and *(5) multi-core awareness* – BFT-SMART architecture takes advantage of the multicore processors present in modern servers to improve some costly processing tasks on the critical path of the protocol. In this section we focus on principles 1, 2 and 3.

All protocols used in BFT-SMART require the partially synchronous system model with $3f+1$ replicas for tolerating Byzantine failures and $2f+1$ if only crashes are considered, just like in similar protocols (PBFT and VR - Section 4.3). Independently of the configuration (crash or BFT), the system requires reliable point-to-point links between processes for communication. These links are implemented using message authentication codes (MACs) over TCP/IP. The symmetric keys for the replica-replica channels are generated through signed Diffie-Helman using a pair of RSA keys per replica. The keys for client-replica channels are generated based on the ids of the endpoints, without the need for clients to hold key pairs.

BFT-SMART uses a replica coordination protocol called Mod-SMaRt, a modular protocol that implements BFT SMR using an underlying consensus primitive (Sousa and Bessani 2012). During normal execution, clients send their requests to all replicas and wait for their replies. Total order is achieved through a sequence of consensus instances, each of them deciding a batch of client requests. Each instance is comprised by three communication steps, just like in PBFT. When only crashes are tolerated, the system needs only $n = 2f+1$ replicas to tolerate f *crash faults* and bypasses one of the steps during the consensus execution, becoming similar to VR/Paxos with fast execution (Section 4.3.1).

Besides the ordering protocol, the system also implements novel state transfer and reconfiguration protocols tolerant to Byzantine faults. These protocols are described elsewhere (Bessani et al. 2013; Bessani et al. 2014).

4.6.2. Architecture and Implementation

A key issue when implementing an efficient SMR library is how to break the several tasks of the protocol in an architecture that is robust and efficient. Figure 4.13 presents the

main architecture of a BFT-SMaRt replica with the threads used for staged message processing.

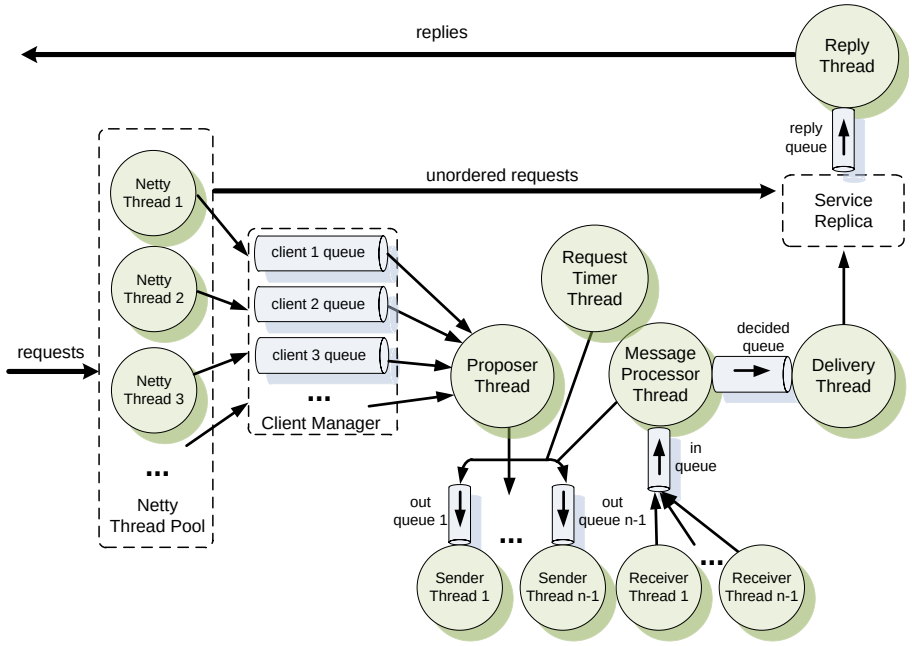

Figure 4.13. BFT-SMaRt replica staged message processing (Bessani et al. 2014).

In this architecture, all threads communicate through *bounded queues*. The client requests are received through a thread pool provided by the *Netty communication framework*. Unordered (read-only) requests are directly delivered to the service implementation. Otherwise, they are delivered to the *client manager* in order to be ordered. Each request is added to the respective client's pending queue.

The *proposer thread* is activated at the leader replica to submit proposals of sequence number assignment for a batch of client requests (Section 4.3). Every message m to be sent by one replica to another is put on the *out queue* to be sent by a *sender thread*. At the receiver replica, a *receiver thread* for this sender will read m and put it on the *in queue*, where all messages received from other replicas are stored in order to be processed.

The *message processor thread* is responsible for dealing with the messages from the SMR protocol (Section 4.3). When a consensus is finished on a replica (i.e., when the request is committed in PBFT parlance), the decided batch is put on the *decided queue* to be delivered, by the *delivery thread*, to the *service replica*, that executes the requests and generates the corresponding replies. The replies are put into the *reply queue* and are sent to the clients by the *reply thread*.

Finally, the *request timer thread* is responsible for triggering view changes (Section 4.3), being periodically activated to verify if some request remained more than a pre-defined time on the pending requests queue.

Overall, the codebase of BFT-SMaRt consists in almost 14000 lines of commented Java code and both the system and its source code can be freely obtained in

`http://code.google.com/p/bft-smart/`.

4.6.3. Performance

This section presents some experimental results from BFT-SMaRt's performance evaluation (Bessani et al. 2014). We present these results here just to give the reader an idea about the performance of SMR systems under different fault models, conditions and workloads. The experiments presented here consist of: (1) some micro-benchmarks designed to evaluate the library's raw throughput; and (2) a performance comparison with some competing systems.

Experimental Setup. All experiments ran with three (CFT) and four (BFT) replicas hosted in separated machines. Up to 1600 clients were distributed uniformly across another four machines. Clients and replicas were deployed in JRE 1.7.0_21 on Ubuntu Linux 10.04, hosted in Dell PowerEdge R410 servers. Each machine have 32 GB of memory and two quad-core 2.27 GHz Intel Xeon E5520 processor with hyper-threading (supporting 16 hardware threads). Machines communicate through an isolated gigabit Ethernet network.

Micro-benchmarks. BFT-SMaRt was evaluated through a set of micro-benchmarks for an "empty" service implemented with BFT-SMaRt to perform raw throughput calculations at the server side. The objective of using such service is to measure the performance of the replication library alone, without applicational overheads. Throughput results were gathered from the leader replica.

Table 4.2 presents approximate values for the peak throughputs of both BFT and CFT setups considering different request/reply sizes (in bytes): 0/0, 100/100, 1024/1024. The table shows that the CFT protocol consistently outperforms its BFT counterpart, what happens due to the smaller number of messages exchanged in the CFT setup.

Workload \ *Configuration*	BFT	CFT
0/0	83000	91000
100/100	72000	80000
1024/1024	16000	21000

Table 4.2. Peak throughput of BFT-SMaRt (in ops/sec) for the CFT and BFT configurations tolerating a single failure and with different workloads (adapted from (Bessani et al. 2014)).

Table 4.3 shows how different request-reply combinations affect the throughput of the system in a BFT setup tolerating a single failure. The results clearly show that large request's payload degrades throughput more than large reply's payload. This can be explained by the increase in the size of the requests' batch proposed for ordering.

These results illustrate the tradeoffs between crash and Byzantine fault tolerance and the influence of different request and reply sizes in the performance of a RSM. These trends appear in most modern SMR systems we are aware of.

Requests \ *Replies*	0 *bytes*	100 *bytes*	1024 *bytes*
0 *bytes*	83801	75138	37320
100 *bytes*	78711	72879	36948
1024 *bytes*	16309	16284	15878

Table 4.3. Peak throughput of BFT-SMaRt (in ops/sec) in a BFT configuration tolerating a single failure for different request and reply sizes.

Comparison with others. BFT-SMaRt was compared (both in BFT and CFT configurations) with PBFT (Castro and Liskov 2002), UpRight (Clement et al. 2009a) and JPaxos (Santos and Schiper 2013) considering the 0/0 benchmark. Table 4.4 shows the *peak sustained throughput* obtained for all these systems and the associated number of clients required to achieve this throughput.

System	*Peak Throughput*	*Clients*	*Throughput 200*
BFT-SMaRt	83801	1000	66665
PBFT	78765	100	65603
UpRight	5160	600	3355
CFT-SMaRt	90909	600	83834
JPaxos	62847	800	45407

Table 4.4. Throughput of different replication libraries for the 0/0 benchmark and $f = 1$. *Throughput 200* reports the throughput with 200 clients.

The results presented in Table 4.4 show that BFT-SMaRt achieves a peak sustained throughput higher than all other systems. The table also presents the throughput of the systems using 200 clients (the maximum supported by PBFT without crashing).

4.6.4. A Key-Value Store using BFT-SMaRt

This section describes the implementation of a dependable main-memory KV-Store based on BFT-SMaRt. The developed datastore should provide an interface similar to the `java.util.Map` interface from the Java API. To simplify the design, we consider keys and values as String objects. We implemented the `put`, `get` and `remove` methods of the aforementioned Java interface together with an additional `list` operation for obtaining the set of keys stored in the system.

Two main classes are used in order to build such service on top of the BFT-SMaRt library. The `ServiceReplica` is used at server side to instantiate a BFT-SMaRt replica while the `ServiceProxy` is used at client side for accessing the replicated service. The instantiation of `ServiceReplica` requires the provision of an unique replica id (which is mapped to an IP and port in a configuration file) and implementations of an `Executable` (which defines the methods called when the service needs to process a request) and a `Recoverable` (which defines the state management methods) interfaces. At the client side, to start a `ServiceProxy` it is necessary only the id of the client. Besides that, both the clients and servers need to have access to the configuration files of the system (which includes the addresses of the replicas).

4.6.4.1. Server Side

The abstract class `DefaultSingleRecoverable` implements both the interfaces `Executable` and `Recoverable` and can be used as the basis for a BFT-SMART service. Using this class, a developer only needs to provide the implementation for the abstract methods described in Table 4.5. BFT-SMART invokes both `executeOrdered` and `executeUnordered` methods upon delivering client requests (or operations) to the replica. Furthermore, BFT-SMART also invokes the state management methods (`getSnapshot` and `installSnapshot`) for obtaining the state of the service when checkpoints are needed and updating the state of a recovered or delayed replica, respectively.

This class was extended by the main class of our KV-Store replica, the`TreeMap-Server`. Listing 4.1 shows the fields of our class and its constructor, which instantiate the `ServiceReplica` and a `TreeMap<String, String>` data structure for holding the data stored in the replica. Starting a server in this example requires the instantiation of this class (not show in this example).

Requests Execution Operations (`Executable` interface)	
public byte[] executeOrdered(byte[] cmd, MsgContext ctx) Parameters: *cmd* – serialized client request; *ctx* – request metadata Returns the serialized reply for ordered execution of *cmd*	implemented for executing ordered requests
public byte[] executeUnOrdered(byte[] cmd, MsgContext ctx) Parameters: *cmd* – serialized client request; *ctx* – request metadata Returns the serialized reply for the unordered execution of *cmd*	implemented for executing unordered requests
State Management Operations (`Recoverable` interface)	
public byte[] getSnapshot() Returns the serialized snapshot of the application state	implemented for creating a snapshot of the application state
public void installSnapshot(byte[] state) Parameters: *state* – the snapshot of the application state to be installed	implement for installing a snapshot of the application state

Table 4.5. Server-side operations to be implemented by a service.

```
1  public class TreeMapServer extends DefaultSingleRecoverable {
2          ServiceReplica replica = null;
3          Map<String, String> table;
4
5          public TreeMapServer(int id) {
6                  replica = new ServiceReplica(id, this, this);
7                  table = new TreeMap<String, String>();
8          }
9          ...
10 }
```

Listing 4.1. Server constructor method.

The next step for implementing the replica is to define its behavior when ordered and unordered requests are delivered. Listing 4.2 shows the code for implementing all operations from the service. As can be seen in the code, our implementation (1) parses the command received to identify the operation (the enum `RequestType` will be defined later on Listing 4.5), (2) reads the appropriate fields of the command, (3) performs

the operation in the data structure and (4) returns the results to the BFT-SMaRt, that forwards it to the client. Notice that we use some stream classes provided by the Java API for marshaling/unmarshaling the transmitted data.

```
public byte[] executeOrdered(byte[] cmd, MsgContext ctx) {
    ByteArrayInputStream in = new ByteArrayInputStream(cmd);
    DataInputStream dis = new DataInputStream(in);
    int reqType;
    try {
        reqType = dis.readInt();
        if(reqType == RequestType.PUT) {
            String key = dis.readUTF();
            String value = dis.readUTF();
            String oldValue = table.put(key, value);
            byte[] resultBytes = null;
            if(oldValue != null){
                  resultBytes = oldValue.getBytes();
            return resultBytes;
        } else if(reqType == RequestType.REMOVE) {
            String key = dis.readUTF();
            String removedValue = table.remove(key);
            byte[] resultBytes = null;
            if(removedValue != null)
                   resultBytes = removedValue.getBytes();
            return resultBytes;
        if(reqType == RequestType.GET) {
            String key = dis.readUTF();
            String readValue = table.get(key);
            byte[] resultBytes = null;
            if(readValue != null)
                resultBytes = readValue.getBytes();
            return resultBytes;
        } else if(reqType == RequestType.LIST) {
            Set keys = table.keySet();
            byte[] keysInBytes = toBytes(keys);
            return keysInBytes;
        } else {
            System.out.println("Unknown request type: " + reqType);
            return null;
        }
    } catch (IOException e) {
        System.out.println("Exception reading data in the replica.");
        return null;
    }
}
```

Listing 4.2. Implementation of executeOrdered method.

For the considered service, the `get` and the `list` operations are read only (they do not change the data structure contents) and thus can be tentatively processed without replica coordination[13]. Therefore, we can execute them even if delivered as unordered operations, as described in Listing 4.3. The processing of these operations follows the same steps of ordered requests.

Finally, the implementation of the state management methods is trivial since it only serializes and gets the service state or deserializes and applies a received snapshot (Listing 4.4).

[13]Interestingly, these operations can also be delivered as ordered requests since the library may order read-only operations in case of concurrency or failures (see Section 4.3.2).

```
public byte[] executeUnordered(byte[] cmd, MsgContext msgCtx) {
    ByteArrayInputStream in = new ByteArrayInputStream(cmd);
    DataInputStream dis = new DataInputStream(in);
    int reqType;
    try {
        reqType = dis.readInt();
        if(reqType == RequestType.GET) {
            //lines 23-28 of Listing 2
         } else if(reqType == RequestType.LIST) {
            //lines 30-32 of Listing 2
        } else {
            System.out.println("Unknown request type:"+reqType);
            return null;
        }
    } catch (IOException e) {
         System.out.println("Exception reading data in the replica.");
         return null;
    }
}
```

Listing 4.3. Implementation of executeUnordered method.

```
public byte[] getSnapshot() {
     try {
       ByteArrayOutputStream bos = new ByteArrayOutputStream();
       ObjectOutputStream out = new ObjectOutputStream(bos);
       out.writeObject(table);
       out.flush(); out.close(); bos.close();
       return bos.toByteArray();
     } catch (IOException e) {
       System.out.println("Exception when trying to take a snapshot.");
       return new byte[0];
     }
}

public void installSnapshot(byte[] state) {
     ByteArrayInputStream bis = new ByteArrayInputStream(state);
     try {
        ObjectInput in = new ObjectInputStream(bis);
        table = (Map<String, String>)in.readObject();
        in.close(); bis.close();
     } catch (Exception e) {
        System.out.print("Exception installing the snapshot.");
     }
}
```

Listing 4.4. Implementation of state management methods.

4.6.4.2. Client Side

At client side, each logical client must instantiate one `ServiceProxy` with a distinct id for representing itself to the replicas. This class implements the SMR client-side protocols and provides the methods described in Table 4.6 to issue commands to the servers.

For our KV-Store client, we aim to implement a simple stub that provides synchronous methods that hide the details about accessing the RSM. This done by implementing a subset of the `Map` interface in our `TreeMapClient` class (Listing 4.5), which can be instantiated in a Java program.

Listing 4.6 shows the implementation of the four methods of the proposed service. Since we are interested only in synchronous methods, we do not use the *invokeAsyn-*

Requests Execution Operations (`ServiceProxy` methods)	
public byte[] invokeOrdered(byte[] request) Parameters: *request* – the serialized request to be sent to the application Returns the serialized reply for ordered execution of *request*	used for invoking an operation that should be ordered
public byte[] invokeUnordered(byte[] request) Parameters: *request* – the serialized request to be sent to the application Returns the serialized reply for unordered execution of *request*	used for invoking an operation that does not need to be ordered
public void invokeAsynchronous(byte[] request, ReplyListener listener, int[] receivers, MsgType type) Parameters: *request* – the serialized request to be sent to the application; *listener* – a callback to receive the replies; *receivers* – the destinations for *request* (usually, all servers); *type* – request type (ordered or unordered)	used for invocating an operation asynchronously, i.e., without blocking waiting for replies

Table 4.6. Client-side operations.

```
1  public class TreeMapClient implements Map<String, String> {
2          ServiceProxy clientProxy = null;
3          enum RequestType {PUT, REMOVE, GET, LIST}
4
5          public TreeMapClient(int clientId) {
6                  clientProxy = new ServiceProxy(clientId);
7          }
8          ...
9  }
```

Listing 4.5. Client constructor method.

chronous operation defined in Table 4.6. Furthermore, it is worth to mention that the implementation of `put` and `remove` use the `invokeOrdered` method of the `ServiceProxy` while `get` and `list` use the `invokeUnordered`.

A step-by-step tutorial showing a complete code for an example similar to this one together with instructions of how to configure and run it can be found in following link: `http://code.google.com/p/bft-smart/wiki/GettingStarted`.

4.7. Final Remarks

This chapter presented a guided tour on the main aspects of the theory and practice of state machine replication. Our main objective through this document was to describe the fundamental aspects and protocols required for implementing state machine replication and present a brief discussion about the recent work and applications related with this technique. We hope this document help researchers and developers to devise reliable services as replicated state machines.

Acknowledgments. We warmly thank Nuno Neves, Miguel Correia, André Nogueira, Pedro Costa and Vinicius Cogo for their feedback in earlier versions of this chapter. We also thank João Sousa and Marcel Santos for their contributions to the BFT-SMART development. We would also like to thank the support by the EC through projects FP7-317871 (BiobankCloud) and FP7-257243 (TClouds), and by the FCT through the SITAN project (PTDC/EIA-EIA/113729/2009) and the LaSIGE Strategic Project (PEst-OE/EEI/UI0408/2014).

```
public String put(String key, String value) {
     ByteArrayOutputStream out = new ByteArrayOutputStream();
     DataOutputStream dos = new DataOutputStream(out);
     try {
         dos.writeInt(RequestType.PUT);
         dos.writeUTF(key);
         dos.writeUTF(value);
         byte[] reply = clientProxy.invokeOrdered(out.toByteArray());
         return (reply != null)?new String(reply):null;
     } catch(IOException ioe) {
         System.out.println("Exception putting value into treemap.");
         return null;
     }
}

public String remove(String key) {
     ByteArrayOutputStream out = new ByteArrayOutputStream();
     DataOutputStream dos = new DataOutputStream(out);
     try {
         dos.writeInt(RequestType.REMOVE);
         dos.writeUTF(key);
         byte[] reply = clientProxy.invokeOrdered(out.toByteArray());
         return (reply != null)?new String(reply):null;
     } catch(IOException ioe) {
        System.out.println("Exception removing value from the treemap.");
        return null;
     }
}

public String get(String key) {
     try {
         ByteArrayOutputStream out = new ByteArrayOutputStream();
         DataOutputStream dos = new DataOutputStream(out);
         dos.writeInt(RequestType.GET);
         dos.writeUTF(key);
         byte[] reply = clientProxy.invokeUnordered(out.toByteArray());
         return (reply != null)?new String(reply):null;
      } catch(IOException ioe) {
         System.out.println("Exception getting value from the treemap.");
         return null;
      }
}

public Set list() {
      try {
          ByteArrayOutputStream out = new ByteArrayOutputStream();
          DataOutputStream dos = new DataOutputStream(out);
          dos.writeInt(RequestType.LIST);
          byte[] reply = clientProxy.invokeUnordered(out.toByteArray());
          ByteArrayInputStream in = new ByteArrayInputStream(reply);
          ObjectInputStream ois = new ObjectInputStream(in);
          return (Set) ois.readObject();
      } catch(IOException ioe) {
          System.out.println("Exception getting the size the treemap.");
          return null;
      }
}
```

Listing 4.6. Implementation of client-side operations.

References

Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005). Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Adya et. al, A. (2002). Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Akkoyunlu, E. A., Ekanadham, K., and Huber, R. V. (1975). Some constraints and tradeoffs in the design of network communications. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 67–74.

Alseberg, P. and Day, J. (1976). A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644.

Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2008). Byzantine replication under attack. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition.

Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. (2011). Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *Proceedings of the ACM International Conference on Management of Data*, pages 1–10.

Bessani, A. (2011). From byzantine fault tolerance to intrusion tolerance (a position paper). In *Proceedings of the Workshop on Recent Advances in Intrusion-Tolerant Systems*.

Bessani, A., Alchieri, E., Correia, M., and Fraga, J. (2008). DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Bessani, A. N., Correia, M., da Silva Fraga, J., and Lung, L. C. (2009). Sharing memory between Byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–443.

Bishop, M. (2002). *Computer Security: Art and Science*. Addison-Wesley.

Bolosky, W., Bradshaw, D., Haagens, R., Kusters, N., and Li, P. (2011). Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

Botelho, F., Ramos, F., Kreutz, D., and Bessani, A. (2013). On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Proceedings of the European Workshop on Software Defined Networks*.

Burrows, M. (2006). The Chubby lock service. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Cachin, C. and Samar, A. (2004). Secure distributed DNS. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Calder et al., B. (2011). Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Chandra, T., Griesemer, R., and Redstone, J. (2007). Paxos made live - An engineering perspective. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*.

Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: A comprehesive study. *ACM Computing Surveys*, 33(4):427–469.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riché, T. (2009a). UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009b). Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

Corbett et al., J. (2012). Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Dixon, C., Uppal, H., Brajkovic, V., Brandon, D., Anderson, T., and Krishnamurthy, A. (2011). Ettm: A scalable fault tolerant network manager. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322.

Elnikety, S., Pedone, F., and Zwaenepoel, W. (2005). Database replication using generalized snapshot isolation. In *Proceeding of the IEEE International Symposium on Reliable Distributed Systems*.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. (2010). Availability in globally distributed storage systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2013). Analysis of OS diversity for intrusion tolerance. *Software - Practice and Experience*. (early view).

Garcia, R., Rodrigues, R., and Preguiça, N. (2011). Efficient middleware for Byzantine fault tolerant database replication. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2009). *Database Systems: The Complete Book (2nd Ed)*. Pearson Education.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Gibson, G., Nagle, D., Amiri, K., Butler, J., Chang, F., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. (1998). A cost-effective, high-bandwidth storage architecture. In *Proceedings of the ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 92–103.

Goldereich, O. (2001). *Fundamentals of Criptography: Basic Tools*, volume 1. Cambridge Press, Cambridge - Massachussets.

Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010). The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR 94-1425, Department of Computer Science, Cornell University, New York - USA.

Herlihy, M. (1991). Wait-free synchronization. *ACM Trans. on Programing Languages and Systems*, 13(1):124–149.

Herlihy, M., Lucangco, V., and Moir, M. (2003). Obstruction-free syncronization: double-ended queues as an example. In *Proc. of the 23th IEEE Int. Conference on Distributed Computing Systems - ICDCS 2003*, pages 522–529.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

Hunt, P., Konar, M., Junqueira, F., and Reed, B. (2010). Zookeeper: Wait-free coordination for Internet-scale services. In *Proceedings of the USENIX Annual Technical Conference*.

Junqueira, F., Bhagwan, R., Hevia, A., Marzullo, K., and Voelker, G. M. (2005). Surviving internet catastrophes. In *Proceedings of the USENIX Annual Technical Conference*.

Junqueira, F., Reed, B., and Serafini, M. (2011). Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., Schröder-Preikschat, W., and Stengel, K. (2012). CheapBFT: resource-efficient Byzantine fault tolerance. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Kapitza, R., Schunter, M., Cachin, C., Stengel, K., and Distler, T. (2010). Storyboard: Optimistic deterministic multithreading. In *Workshop on Hot Topics in System Dependability*.

Kapritsos, M. and Junqueira, F. P. (2010). Scalable agreement: Toward ordering as a service. In *Workshop on Hot Topics in System Dependability*.

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about Eve: Execute-verify replication for multi-core servers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Kirsch, J., Goose, S., Amir, Y., Wei, D., and Skare, P. (2013). Survivable SCADA via intrusion-tolerant replication. *IEEE Transactions on Smart Grids*, 4(3).

Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. (2010). Onix: A distributed control platform for large-scale production networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39.

Lamport, L. (1978a). The implementation of reliable distributed multiprocess systems. *Computer Networks*, 1(2):95–114.

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

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lamport, L. (2006). Fast paxos. *Distributed Computing*, 19(2).

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41(1).

Lamport, L., Shostak, R., and Pease, M. (1982). The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Lee, E. K. and Thekkath, C. A. (1996). Petal: Distributed virtual disks. In *Proceedings of the ACM Int. Conference on Architectural Support for Programming Languages and Operating Systems*.

Liskov, B. and Cowling, J. (2012). Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT.

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L. (1991). Replication in the Harp file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Littlewood, B., Popov, P., and Strigini, L. (2001). Modeling software design diversity: A review. *ACM Computing Surveys*, 33(2):177–208.

Lorch, J., Adya, A., Bolosky, W., Chaiken, R., Douceur, J., and Howell, J. (2006). The SMART way to migrate replicated stateful services. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kauffman.

Mao, Y., Junqueira, F. P., and Marzullo, K. (2008). Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

Martin, J.-P. and Alvisi, L. (2006). Fast Byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215.

Miller, R. (2008). Explosion at The Planet causes major outage. *Data Center Knowledge*.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Obelheiro, R., Bessani, A., Lung, L., and Correia, M. (2006). How practical are intrusion-tolerant distributed systems? Technical Report DI/FCUL TR 06-15, University of Lisbon.

Oki, B. M. and Liskov, B. (1988). Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 8–17.

Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., Kim, C., and Karri, N. (2013). Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013*.

Powel, D. (1996). Group communication. *Communications of the ACM*, 39(4):50–53.

Rao, J., Shenkita, E. J., and Tata, S. (2011). Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*.

Santos, N. and Schiper, A. (2013). Achieving high-throughput State Machine Replication in multi-core systems. In *Proceeding of the IEEE International Conference on Distributed Computing Systems*.

Schiper, A., Birman, K., and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Sousa, J. and Bessani, A. (2012). From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceeding of the European Conference on Dependable Computing*.

Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2010). Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465.

Sousa, P., Neves, N. F., and Verissimo, P. (2007). Hidden problems of asynchronous proactive recovery. In *Proceedings of the Workshop on Hot Topics in System Dependability*.

Verissimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*.

Veronese, G., Correia, M., Bessani, A., Chung, L., and Verissimo, P. (2013). Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1):16–30.

Veronese, G. S., Correia, M., Bessani, A., and Lung, L. C. (2010). EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proc. of the IEEE International High Assurance Systems Engineering Symposium*.

Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2009). Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceeding of the IEEE International Symposium on Reliable Distributed Systems*.

Wang, Y., Alvisi, L., and Dahlin, M. (2012). Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the USENIX Annual Technical Conference*.

Wood, T., Singh, R., Venkataramani, A., Shenoy, P., and Cecchet, E. (2011). ZZ and the art of practical BFT replication. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*.