Dynamic Adaptation of Byzantine Fault Tolerant Protocols

Carlos Eduardo Alves Carvalho

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor:
Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

October 2017

Acknowledgements

Firstly, I thank my parents, from the bottom of my heart, for making this journey possible and for always being there through thick and thin. I also thank Diana for her caring and teachings. I thank my friends, specially Miguel and Beatriz, for being patient with me and making this journey a delightful one. I would also like to thank Dr. António Venâncio for all of the meta-discussions about this project. I also would like to acknowledge my supervisor, Luís Rodrigues, for welcoming me as his student and for the endless support and sharing of knowledge. Last but not least, I thank all my colleagues at INESC-ID, namely Manuel Bravo, Diogo Barradas, Daniel Porto and Nuno Duarte for their support and friendship.

To each and every one of you – Thank you.

Lisbon, October of 2017

Carlos Eduardo Alves Carvalho

For my mother,

Resumo

O problema do consenso distribuído na presença de faltas bizantinas tem recebido particular atenção nas últimas décadas. Em parte devido ao enfraquecimento da confiablidade dos sistemas físicos e em parte devido ao aumento do número de ataques maliciosos. Assim, existem hoje diversos protocolos para este efeito, cada um otimizado para condições de execução particulares. Uma vez que na maioria dos casos os sistemas reais operam em condições dinâmicas, importa desenvolver mecanismos que permitam adaptar os protocolos em tempo de execução ou substituir um protocolo por outro mais adequado às condições correntes.

O problema da adaptação dinâmica de protocolos de consenso não é novo, mas a literatura é escassa para o caso bizantino e não existem trabalhos que permitam comparar as soluções existentes. Este trabalho tem dois objetivos complementares. Em primeiro lugar, estuda como as diferentes técnicas de adaptação dinâmica propostas para o modelo de falta por paragem podem ser aplicadas na presença de faltas bizantinas. Em segundo lugar, através da concretização destas técnicas numa moldura de software comum, baseada no pacote de código aberto BFT-SMaRt, apresenta um estudo comparativo do desempenho das mesmas.

Abstract

The problem of distributed consensus in the presence of Byzantine faults has received particular attention in recent decades. Today a variety of solution to this problem exist, each optimized for particular execution conditions. Given that, in most cases, real systems operate under dynamic conditions, it is important to develop mechanisms that allow the algorithms to be adapted at runtime or to switch between different algorithms so that is possible to optimize the system to the current conditions.

The problem of dynamic adaptation of consensus algorithms is not new, but the literature is scarce for the Byzantine case and there is no comprehensive comparison of existing solutions. This work has two complementary objectives. First, it studies how the different dynamic adaptation techniques proposed for the crash failure model can be applied in the presence of Byzantine faults. Second, it presents a comparative study of the performance of these switching algorithms in practice. For that purpose, we have implemented the switching algorithms in a common software framework, based on the open source BFT-SMaRt package. Using this common framework we have performed an extensive evaluation that offers useful insights on the practical effects of different mechanisms used to support the run-time switching among Byzantine protocols.

Palavras Chave
Keywords

Palavras Chave

Tolerância a Faltas Bizantinas

Adaptação Dinâmica

Protocolos de Consenso Adaptáveis

Protocolos de Consenso Bizantino

Replicação de Máquinas de Estado

Keywords

Byzantine Fault Tolerance

Dynamic Adaptation

Adaptable Consensus Protocols

Byzantine Consensus

State Machine Replication

Contents

List of Figures

List of Tables

Acronyms

SMR	State Machine Replication
BFT	Byzantine Fault Tolerant
CFT	Crash Fault Tolerant
MAC	Message Authentication Code
SHA-256	Secure Hash Algorithm with 256-bit Output

1 Introduction

This dissertation addresses the problem of dynamic adaptation of Byzantize Fault Tolerant (BFT) consensus protocols. It departs from a comprehensive analysis of the related work, namely of previous techniques to perform dynamic adaptation of consensus protocols in different fault-models (including Crash Fault Tolerant (CFT)). Subsequently, it discusses how these solutions can be adapted to the BFT case, in order to derive a catalog of adaptation strategies that can be used in different contexts. Finally, it offers a comparison of the performance of these techniques in practice, by experimentally evaluating an implementation of the algorithms in a common framework.

1.1 Motivation

State Machine Replication (SMR)(Schneider 1990) is one of the fundamental techniques for providing fault tolerance. At its core, this technique uses a distributed consensus algorithm so that all the replicas can agree in the order in which they should process the requests. This work focuses mainly on the case where one intends to use SMR to tolerate Byzantine faults (BFT). This is motivated by the observation that Byzantine faults tend to be more likely in the present days as electromagnetic phenomena is potentiated by the trends in chip development and malicious cyber-attacks are ever more frequent.

Among the systems that have been proposed to accomplish BFT state machine replication we highlight PBFT (Castro, Liskov, et al. 1999), Aardvark (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009), and Zyzzyva(Kotla, Alvisi, Dahlin, Clement, and Wong 2007). Each of these systems operates better under certain conditions, and worse in others, with none surpassing all others in all situations, as shown by Singh, Das, Maniatis, Druschel, and Roscoe (2008). Zyzzyva performs better when no faults occur and the network is stable. On the other hand, when faults frequently occur, Aardvark operates better than the rest, sacrificing performance in the fault-free case. In addition, the performance of the PBFT is less sensitive to the increased size of the messages exchanged when compared to the Zyzzyva. These differences motivate the interest of switching among different algorithms, or to dynamically adapt a given algorithm.

Therefore, in this work we are interested in the study of mechanisms that allow to adapt, or to replace, in runtime, a consensus algorithm for another. This is relevant since most of the practical applications of SMR are subject to variations of their execution environment, from changes in the load imposed by clients to variations on the network performance. Furthermore, as there is no one-size-fits-all algorithm solution for a range of extended operating conditions, the only way to ensure a good performance in face of a variable envelope is to perform dynamic adaptation.

The problem of dynamic adaptation of consensus algorithms is not new and has been well studied for the crash fault model (e.g. (Couceiro, Ruivo, Romano, and Rodrigues 2015), (Mocito and Rodrigues 2006), (Chen, Hiltunen, and Schlichting 2001)). However, the literature is scarce for the Byzantine case and, in fact, several of the previously proposed mechanisms may fail in face of Byzantine faults and need to be modified to operate in such a scenario. Even among algorithms developed taking Byzantine faults into account there is, as far as we know, no work comparing their performance. In this way, those who seek to support dynamic adaptation while tolerating Byzantine faults do not have at their disposal concrete data in order to choose the adaptation technique that best suits the characteristics and objectives of the target system.

Thus, this work has two main goals. First, it studies how the different dynamic adaptation techniques that were previously proposed for the crash failure model can be adapted to work in the presence of Byzantine faults. Second, it presents a comparative study of the performance of these switching algorithms in practice. For that purpose, we have implemented the switching algorithms in a common software framework, based on the open source BFT-SMaRt library (Bessani, Sousa, and Alchieri 2014). Using this common framework we have performed an extensive evaluation that offers useful insights on the practical effects of different mechanisms used to support the run-time switching among Byzantine algorithms.

1.2 Contributions

This work studies, implements and evaluates techniques to perform on-the-fly adaptations on BFT consensus protocols with the goal of offering a better overall performance to a State Machine Replication (SMR) system. This thesis presents then the following contributions:

- A common conceptual framework to rationalize about different techniques for performing dynamic adaptation of BFT consensus protocols.
- A portfolio of techniques for the dynamic reconfiguration of BFT systems, that extends previous techniques proposed for different fault models.
- New insights on the relative performance of these techniques in different operational scenarios.

1.3 Results

The results produced by this thesis are enumerated as follows:

- The design and implementation of several modules and extensions to allow the BFT adaptation of the BFT-SMaRt framework.

- The development and implementation of five different adaptation approaches, in BFT-SMaRt, which implement the support for on-the-fly switching between different consensus protocol behaviours, as well as the adaptation of some configuration parameters of such protocols. This five approaches are combinations of three distinct adaptation algorithms with some possible optimizations.
- An experimental comparative evaluation of the implemented techniques, with regard to its performance in different use cases and conditions of the execution environments.

1.4 Research History

This work was developed as part of the Abyss project. The main goal of the project is to develop and evaluate a complete solution to support dynamic adaptation of BFT protocols. Other team members are working on different modules of the Abyss architecture, such as a BFT policy-driven adaptation manager (to issue the adaptation given the current environment) and a BFT sensor infrastructure to capture the environment variables. An early prototype of the Abyss architecture has been developed by Frederico Sabino (2016); this works extends that previous protocol adaptation technique in an significant manner.

Initially this work was focused not only in developing on-the-fly protocol switching, but also on other adaptations, namely leader switching, batch size adaptation, among others. During the development of the work the aim focused more on the protocol switching, because it was the more general approach, making no assumption of the actual protocol running and subsuming all other adaptations.

This thesis greatly benefited from the fruitful discussions with Miguel Pasadinhas about the usability of the system from the perspective of a adaptation manager implementation; Professor Alysson Bessani and João Sousa about the operation of BFT-SMaRt; Daniel Porto and Manuel Bravo about conceptual and technical aspects of this work.

A subset of the work reported in this dissertation has been published in Actas do Nono Simpósio de Informática, INForum 17 (Carvalho, Porto, Rodrigues, and Bessani 2017). The paper has been awarded with the prize for the best student paper.

This work was developed at INESC-ID and LaSIGE and has been supported in part by FCT through projects PTDC/EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013.

1.5 Structure of the Document

The remaining of this document is organized as follows. Chapter 2 provides a brief introduction to BFT systems and presents several adaptation techniques described in the literature. Chapter 3 describes a conceptual model to compare the distinct adaptation techniques, discusses the changes needed to apply such techniques in a BFT context and presents some possible optimizations. It also describes the architecture and implementational details of the application of such techniques and optimizations in the BFT-SMaRt framework. Chapter 4 presents an experimental comparison and evaluation of the developed adaptation approaches. Lastly, Chapter 5 concludes this document by summarizing the main findings of this work and discussing some directions for future work.

2 Related Work

This chapter starts by briefly introducing the concepts of State Machine Replication (2.1) and Byzantine Fault Tolerance (2.2). Then, some existing BFT SMR systems are described and compared (2.2.2 to 2.2.6). In section 2.3 are presented and discussed some adaptation techniques present in the literature.

In the literature, the term reconfiguration is used sometimes to refer the adaptation of the replicas already deployed, and sometimes to refer to the change in the membership of the set of replicas participating in the consensus. In this work, we use reconfiguration to refer to any change to a configuration parameter or behaviour in the system (including changing the running protocol or the set of deployed replicas). Thus, the terms reconfiguration and adaptation are used interchangeably.

2.1 State Machine Replication

A general approach to provide fault-tolerance is to use State Machine Replication (Schneider 1990). This approach sees the whole system as a finite state machine with a given *state* and that is capable of processing certain *commands*, which modify the state (deterministically). This state is replicated among all replicas of a given service and commands are executed in all of them. The rationale is to have several copies of a server, so if some fail, others can still provide service. It is common to have a *log* with the history of the requests processed by the machine. This serves two purposes: firstly to transfer state if a new replica is initiated (possibly to replace a faulty one); secondly, to enforce synchronization among replicas. As faults may cause some deviations among the state of replicas, the history of each replica is used to decide upon a common history, so that the overall SMR keeps consistent. To decide on the order of the commands executed, there is the need for an agreement among the replicas, that is called *consensus*. A high level representation of the general architecture of a replica in SMR is presented in Figure 2.1.

The replicas participating on a SMR system may change over time, because faulty replicas may be replaced by new ones, or the total number of replicas is changed (e.g. increased to withstand more faults). It is essential that every replica knows about the others in order to carry out consensus: the state about how many and which replicas are participating in consensus is denoted a *view*.

SMR systems can be built to tolerate Crash or Byzantine faults. Both approaches are discussed in more detail in the following sections.

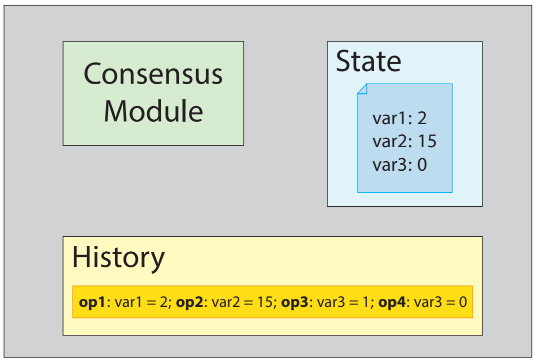


Figure 2.1: The general architecture of a replica in State Machine Replication

2.1.1 Raft

Raft(Ongaro and Ousterhout 2014) is a protocol used to implement SMR in synchronous systems subject to crash faults. Raft can be seen as a more understandable consensus approach than the most known Paxos protocol, originally proposed by Leslie Lamport(Lamport et al. 2001). Paxos relies on three roles: proposers, acceptors and learners. The proposer is responsible for electing a value to be decided with a given sequence number and forwarding this decision to the acceptors. The acceptors receive this chosen value and promise to one another that they will not accept a different decision for that sequence number. When at least $f+1$ acceptors made the promise, then it is ensured that even if f fail, the decision will not be lost. Finally this decision is broadcast to all learners, who may act upon it. Paxos requires at least $2f+1$ replicas in order to tolerate up to f faults, as does Raft.

Raft, as Paxos, is a *leader-based* protocol; one of the replicas is elected to play a special role and act as a coordinator for the protocol, called leader and similar to the Paxo's proposer. The protocol includes the required mechanisms to replace a failed leader by another replica and to make sure that no inconsistency is generated even if, due to the asynchrony of the system, more than one replica believes to be the leader at a given point in time.

The leader replica receives requests from the client and broadcasts them to all the other replicas. Before a request is processed, the protocol ensures that it is ordered and added to the log of, at least, $f+1$ replicas. When a given entry is appended to the log of a majority of the replicas, we say that it is *committed* and it is safe to answer to the client because, despite failures, that operation will always be part of the history of the state machine. Known to be committed, it can be executed and a response returned to the client.

If the leader fails, a new leader is elected. Each replica has a given time-out threshold to wait for messages from the leader. When a time-out occurs, the replica suspects that the leader may have crashed, so it initiates a leader election. Firstly it proposes itself as a candidate for leadership to all other replicas. Then, all replicas cast their vote, and if, and only if, a majority vote for it, it starts to be the leader and informs every other replica of its new role. Every time an election is started, it is started a new *term*, which is sent along all communications. Terms have increasing numbers and are used to enable replicas to know if other replica is further advanced or far behind, as the system progresses. As an example, if a candidate to leadership receives a request for processing a command from another leader and the term is greater than its own term, then it knows that the new leader already won an election further on the progress of the machine. Knowing this, the candidate stops the election, as it knows there was an election "further in the future" in which he did not participate, although a majority of replicas did. This can happen, for example, due to a transient fault in the network, where some packets were dropped, and the replica missed some communications.

When a leader is elected, it is possible that inconsistencies of logs among the replicas may arise. As an example, the old leader may have entries in his log that were not fully replicated yet. To solve this, Raft forces all replicas to replicate the new leaders log. This is done by finding the latest point where the log of the replica is equal to the leader, removing further entries (that not exist in the leader's log), if they exist, and appending the missing entries that are present in the leader's log. Unfortunately, if no preventive measures are taken, the new leader can be a replica that failed to append some commands to his log. This could carry the risk of erasing parts of history which a client already knows of, and thus breaking correctness, because the overall state of the machine would not be consistent with the requests made by client. To solve this, Raft limits who can be elected as leader: only a candidate that has a log with, at least, the same entries as a majority of the replicas can be elected. This ensures that the new leader has all the committed entries, as by design, an entry is committed if a majority of the replicas have it in their logs.

As the main focus of this work is on BFT, we will not present more crash fault tolerant systems, as the on described earlier already provides a general view about the mechanics of crash fault tolerant SMR. In the next section we will present the problem of Byzantine fault tolerance, some systems that solve this problem, and discuss the key differences between crash and Byzantine fault tolerant, as well the differences and similarities among BFT SMR approaches.

2.2 The Byzantine Generals Problem

The Byzantine fault tolerance problem was first described by Lamport, Shostak and Pease in (Lamport, Shostak, and Pease 1982). The authors present the problem starting point as follows: there are several of the Byzantine army camped outside an enemy city, being each division commanded by a general. They must agree on how to perform the attack, because if they don't attack in accordance to each other they might face defeat. The problem arises from the existence of traitors among the generals that might try to impair this agreement so that the Byzantine army fails. Of course, this is a metaphor for a distributed system where some machines may not act like specified or intended.

In the paper a solution, among others, was presented that became the starting point for most practical implementations of BFT SMR. The ultimate objective of the protocol is to have all the non-faulty nodes agreeing on a value. These nodes are the replicas, when we talk about SMR. The value to be agreed upon is suggested by the leader, a node with this special role.

As discussed in the paper, the ability of faulty nodes to lie about the messages received from other nodes introduces difficulty in solving the problem. So as to mitigate this, the proposed solution uses signed messages to prove the source of a given message. This protocol assumes a fully connected, synchronous, network, at most *f* faulty nodes and at least $2f + 1$ nodes. Below, we present an informal description of the solution:

1. Every node starts with a vector $V_i = \emptyset$, representing the collection of values received from other nodes.
2. The leader proposes a value *v*, sending a signed message to all other nodes.
3. Upon receiving a message each node acts accordingly with one of the next cases (in other cases the messages are ignored):
   A) If it receives a message from the leader for the first time, the node adds *v* to V_i. Then it signs the message and resends it to all other nodes.
   B) If the node receives a message with a value v', signed by $k + 1$ nodes (including the leader) and $v' \notin V_i$, the node adds v' to V_i. Then if $k < f$, it signs the message and sends it to all nodes apart from those that already signed the said message, to try to guarantee that, even if f nodes fail, at least one correct node will know about the value proposed by the leader.
4. When a node will not receive more messages, it chooses an action, based on a deterministic function, taking in account the values in V_i.

To know when a node will not receive further messages, it is needed to have some time-out mechanism. To achieve this, it is necessary to assume some maximum time for processing and transmitting a message, therefore a synchronous network needs to be assumed. Although this may seem a flaw, there is no possible solution for solving the problem under an asynchronous network assumption, as the FLP Impossiblity Theorem (Fischer, Lynch, and Paterson 1985) proves. Nevertheless, most recent solutions assume networks where asynchrony may happen during limited periods of time and need for more replicas to ensure safety, which is discussed further on this chapter.

2.2.1 Tangaroa

In this subsection we will present an adaptation of Raft that tolerates Byzantine faults, called Tangaroa (Copeland and Zhong 2014). We will compare both approaches so the differences between crash fault tolerant and Byzantine fault tolerant systems become evident.

The first key difference is that to tolerate Byzantine faults there is a need for more replicas, at least, $3f+1$, to tolerate f faults, as opposed to $2f+1$ in crash fault tolerance (Bracha and Toueg 1983), when dealing with partially synchronous networks [1].

Secondly, it is necessary to ensure authenticity in communications, as replicas may lie about messages received from other replicas. To enforce this, Tangaroa uses digital signatures. As an example, a leader could modify a command received from a client, tricking other replicas to execute something different from the real command and harming the safety of the system. If the client signs its messages, then it is theoretically impossible for a leader do tamper with it (Diffie and Hellman 1976).

Thirdly, a Byzantine leader could starve the system by ignoring clients' requests while continuing to send heartbeats, this is, sending messages proving it has not crashed, although never broadcasting the requests. So there must be a mechanism that is able to detect this and act upon it. This approach solves it by allowing clients to denounce a leader if it does not answers to requests timely, so a leader change is triggered. Nevertheless, another common solution is to have the clients broadcast all of their requests to all the operating replicas.

Furthermore, any replica could lie about its log, when asked to replicate the leader log, compromising the state of the machine. They could state that they had replicated it, but when in reality they have another sequence of requests in its log. So, further proof must be gathered to convince other replicas that a replica has, in fact, all the entries in its log. To prove this, a replica hashes its log and signs it, so others can compare the hashes with their own to check if the histories match.

Another issue arises because a replica could lie and elect himself as leader, without winning an election. To do this, a replica could just broadcast a message stating “I'm the new leader”, making others look to it as the new leader. To prevent this, a recently elected leader, when informing the system of its new role, must sent cryptographic proof that a majority of replicas voted for him (i.e. the signed votes).

A Byzantine leader in Raft, as it is the single node that coordinates what is committed, could also tell that a given entry was committed even if a majority of the replicas had not appended it to their logs. By opposition, in Tangaroa, the responsibility of marking entries as committed is removed from the leader and belongs to all replicas. When a replica appends some entry to the log it broadcasts its action to all other replicas. The replicas collect these messages and identify by themselves that an entry was committed when a majority of replicas has, in fact, appended it.

Finally, in Raft a Byzantine replica could always be proposing elections, sending the system to a loop of infinite elections, where no progress is made. To mitigate this, in Tangaroa, a replica only casts a vote for a new leader if it also suspects that the leader is faulty. Otherwise, the election will be ignored, because all correct replicas will ignore the election proposal.

We can see that some overheads arise when a Byzantine fault tolerant approach is used. There

[1] Discussed in more detail in 2.2.2

is a need for more replicas, which sometimes is an issue, because it makes the whole system more expensive. Moreover there is the need for producing cryptographic proofs to ensure authenticity, which causes a significant overhead in CPU and the time consumed to process communications, both when producing and verifying those proofs. Finally, a need for all-to-all communication and a need for extra communication steps arise, which increase the number of messages in the network exponentially, what can possibly introduce more latency on communications, as well, it demands further effort by the replicas to process all the messages. However, despite this extra overhead, sometimes a Byzantine environment must be assumed, when having a Byzantine failure is not tolerable, for example in critical control software.

Below we present and discuss some approaches for solving BFT SMR in practice.

2.2.2 PBFT

PBFT (Castro, Liskov, et al. 1999) is a widely-studied BFT protocol that aims on solving BFT SMR in a practice. It was the first presented BFT protocol that relaxed the synchrony assumption, which is often not present in real world systems. It does not rely on synchrony to ensure safety, instead, it relies on the assumption that the network has some times of synchrony to ensure liveness (assuring liveness and safety under total asynchrony is impossible (Fischer, Lynch, and Paterson 1985)). Furthermore, some optimizations were also introduced to try to reduce the response time.

As Tangaroa, this approach needs the theoretical minimum of replicas to work, which is $3f + 1$ replicas, where f is the maximum number of faults tolerated. On a high-level view, the protocol works as follows:

1. The client sends an operation request to the leader, that is responsible for ordering the operations on the system.
2. The leader atomically multicasts the request to the other replicas, called backups.
3. All the replicas process the request and answer to the client.
4. The client verifies if it received at least f+1 equal replies, if so it assumes that reply as the result of the operation.

In this system the view states what replica is the leader. When the leader is suspected to be faulty, a view change is carried out and another replica becomes the leader. This suspicion is arose when a backup notices that a request it knows of is taking too long to be executed.

The atomic multicast protocol is composed of three phases: pre-prepare, prepare and commit. The first two, together, guarantee that the requests are totally ordered within a view, even in the presence of a faulty leader. The latter two, in conjunction, guarantee that the ordering is kept among views.

In the pre-prepare phase the leader assigns a sequence number n to the request and sends a PRE-PREPARE message to all replicas. A backup accepts this message if:

- is in the same view as the leader,
- has not accepted a pre-prepare with the sequence number n for a different request, in that view,
- verifies the authenticity of the message.

If a backup accepts the PRE-PREPARE, it enters in the prepare phase. As it does so, it sends a PREPARE message to all other replicas. Every replica registers the PRE-PREPARE messages, its own PREPARE, as well as the PREPARE messages received from other replicas (as long as they have correct signatures and are in the same view).

When a replica has at least $2f$ PREPARE messages from other backups that are coherent with the PRE-PREPARE, it multicasts a COMMIT message to all other replicas. When a a replica receives has at least $2f + 1$ COMMIT messages, possibly including its own, matching the PRE-PREPARE, it processes the operation requested by the client and replies to it.

A view-change is initiated when a backup notices that a request is taking to long to be processed. To change the view, the replica stops participating in the processing of requests and multicasts a view-change message. When the leader of the new view gets $2f$ view-change messages from other replicas, it informs all backups of the new view (with proof that $2f + 1$ replicas agreed upon that).

In order to make the system faster in practice, among other optimizations, this paper introduces the usage of Message Authentication Code (MAC) instead of digital signatures to reduce the load on the CPU of the replicas. MACs are computationally less expensive because they use symmetric cryptography, as opposed to the asymmetric cryptography used to produce digital signatures, which needs more complex mathematical computations to be carried out.

2.2.3 Zyzzyva

Zyzzyva aims to make a fast BFT SMR system by using speculation. In this approach, a request is executed without running an agreement among the replicas to order it, in the hope that no failures occur, unlike other systems such as PBFT. Logically, to ensure safety even under the presence of Byzantine faults, an agreement is run when suspicions of faulty behaviour occur. To detect this type of behaviour, the client is responsible to detect and inform about some divergence of answers received by it. As PBFT, Zyzzyva relies on a leader to give a sequence number to the requests and uses $3f + 1$ replicas (f being the total number of tolerated faults).

In the fast case, where there are no faults nor divergence on the state of the replicas, the protocol works as follows:

1. The client sends a request to the leader.

2. The leader gives a sequence number to the request and forwards it to all the replicas.

3. Each replica executes (speculatively) the request and sends the response to the client. If the client receives $3f + 1$ equal answers, it is safe to rely on the answer because all the correct replicas will keep this request consistently ordered in their history.

On the other hand, if the client receives only between $2f + 1$ and $3f$ matching answers, further steps must be taken to ensure safety. It may indicate that the state of the replicas is diverging, due to faults in them or the network, because there is no proof that all processed the request the same way. Therefore, the consistency of the general state of the system may be at risk of becoming inconsistent when a view change happens. This could happen, for example, under the presence of a faulty leader, that could orchestrate an attack by lying to some correct replicas, making it possible to agree on a faulty state when a view-change happens. To prevent this kind of faulty behaviour to harm the system's correctness, the client builds and distributes a certificate that proves that at least $2f + 1$ replicas agree on the answer of the request with a given sequence number, so when a view change happens, it is ensured that there is proof that a quorum of $2f + 1$ replicas agreed on the order of the request in the past. To ensure that enough servers received this proof, it awaits the acknowledgement of at least $2f+1$ replicas. This amount of replicas ensures that even the presence of f faulty nodes that will lie about this acknowledgement, there is a sufficient number of replicas ($f + 1$) to prove that this agreement existed.

If the client receives less than $2f + 1$ matching responses, which is not enough to ensure that a majority of correct replicas agreed on some response, it suspects a faulty leader and resends the request to all replicas. If a given replica has not processed that request already, it forwards the request to the leader. If, after a certain amount of time, it has not received the corresponding request ordered by the leader, the replica starts a view-change.

The usage of speculation and having an agreement protocol with just two steps introduces extra complexity on the view-change protocol. Having just two phases, omitting a phase when the replicas share their state, makes the traditional view-change protocol unsafe, as the correct replicas might not be able to initiate a view-change in the presence of a faulty leader (it is possible that only f correct replicas notice it). To ensure that at least $2f$ replicas commit to a view-change, there is a need for an extra step in the view-change protocol, where the replicas state their disagreement about the leader. So, the protocol works as follows:

1. A replica informs all other about is suspicion about the leader.

2. Upon receiving $f + 1$ confirmations of the suspicion on the leader from other replicas, it is created a proof that at least one correct replica suspects the leader and it is sent along a view-change message. All correct replicas, given this proof, will commit also to the view-change.

3. The leader of the new view collects $2f + 1$ view-change messages and then sends a new-view message with proof of that.

4. Finally, when a replica receives the new-view, it changes its view.

2.2.4 Aardvark

Aardvark (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009) addresses the problem of BFT SMR with a very different mindset of the previous solutions. The authors reject every optimization that could impair the performance in cases where faults happen. This way, Aardvark maintains a steady throughput, even in the presence of faults, making it more robust w.r.t. Byzantine behaviour, in the replicas and the clients. This system relies on a similar communication pattern to PBFT, using a leader to sequence clients' requests and a three phase agreement protocol, although it presents some key differences on the implementation.

Firstly, to minimize the harm a Byzantine client could make on the system, clients digitally sign their requests, instead of using MAC (although, MAC are also used as an optimization in some cases). This way it is ensured that if a replica can prove the authenticity of a request, all other can too, because signatures provide non-repudiation, so when a client signs a message, there is (theoretically) undeniable proof that it indeed sent that message. Although signatures are more expensive to compute than MAC, this simplifies the protocol, removing some corner cases found on other systems. For example, the authors found out that in PBFT and Zyzzyva if a client would send a request with a valid MAC for the primary and invalid MACs for the other replicas, it would render the system unusable. PBFT would incur in recurring view changes, while Zyzzyva would invoke a conflict resolution protocol that in practice, due to not be fully implemented, would never finish.

Secondly, also to mitigate the impact of faulty clients, it uses separate queues for messages from the clients and replica-to-replica communication. This way, a replica can guarantee that only a portion of the resources can be used by the client, allocating only part of the computation time to process client's requests. This way a client can not render a replica unusable for participating in the ordering of requests by flooding its queue with requests. This approach also uses independent network interface controllers and wires to link each pair of replicas. So, at the expense of having to rely on point to-point communication, it is possible to receive messages in parallel and also shut down links to faulty nodes trying to develop a denial of service attack.

Lastly, Aardvark carries out view-changes regularly, as opposed to carrying it only as a last-resort measure. The authors justify this decision stating that the cost of having a faulty leader surpasses the cost of executing periodic view-changes. In order to carry the changes periodically, the replicas demand an increasing minimum throughput of requests by the leader, as soon as the leader fails to provide such throughput, the replicas start a view-change.

2.2.5 Fast Byzantine Consensus

Martin and Alvisi present a Byzantine consensus protocol. Fast Byzantine Consensus (Martin and Alvisi 2006), capable of finishing in just two steps of communication, like the one present in Zyzzyva, but without speculative executions. However, to achieve this, it uses $5f + 1$ replicas to tolerate up to f faults.

This work does not describe a full implementation of state machine replication, nevertheless it is easy to derive such system from the consensus behaviour.

As the other discussed solutions, this ordering protocol ensures safety in an asynchronous network and liveness when synchrony is achieved. It also tolerates, in theory, an infinite number of clients, including those who present Byzantine behaviour. To tolerate f faults of processes playing each role, it demands n replicas such that they can accommodate at least $3f+1$ proposers, $5f+1$ learners and $3f+1$ acceptors, where each replica can play any number of this roles.

When there is only one correct leader proposing, then the protocol completes in two steps:

1. The leader proposes a value, sending it to all acceptors;
2. The acceptors accept this value and inform the learners of it. Learners accept such value when they observe that, at least, $(a+f+1)/2$ [2] accepted such value.

In order to tolerate asynchrony in the network, every replica retransmits the requests performed until they receive a confirmation response. Moreover, to detect a faulty leader, learners inform not only proposers of the learned values, but also acceptors. This way, if an acceptor doesn't hear from enough learners within a given time span, it raises an accusation against the leader. If a quorum of proposers suspect the leader, than a new leader is elected. The leader election does not deviate considerably from the leader election of other already discussed protocols, the new leader is deterministically determined and builds a proof of the current state of execution, which delivers to all acceptors.

A variation of this protocol is also presented, called Parametrized FaB Paxos, which flexibilizes the number of replicas needed to unsure correctness, standing between the typical $3f+1$ with three communication steps, and the faster two-step consensus using $5f+1$. This variation demands $3f+2t+1$ replicas to ensure safety up to f faults and guarantees two-step execution with up to t faults.

2.2.6 Discussion

When comparing the machinery needed to tolerate Byzantine faults versus the mechanisms needed to tolerate only crash faults, it is evident that the first carries significant overheads. Firstly there is a need for, at least, more f replicas to tolerate f faults. Secondly, all-to-all communication introduces much more load in the network, which can cause starvation. Moreover, the use of authenticated messages introduces more load on the CPU to process the cryptography, as increases the time needed to process each message. Nonetheless, this all-to-all communication, with a heavy CPU load to calculate authentication proofs, can be avoided in some cases as show by Chain in (Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015). In Chain all replicas form a chain and a request is transferred from node to node in the chain, reducing the cost of all-to-all authenticated communication, but introducing extra latency,

[2] a is the total number of acceptors in the system and f is the number of tolerated faults.

because the messages are processed sequentially in every nodes, instead of in parallel. Moreover, despite not being fully experimented in a SMR application, Fast Byzantine Consensus promises a better performance than other discussed protocols, has it only needs two steps to finish in most cases and does not suffer from the drawbacks of speculative execution like Zyzzyva. Actually, it can be optimized to finish tentatively in just one step in the SMR case. Nevertheless, it is more resource consuming than CFT consensus protocols. However, as stated earlier, these overheads in latency, network resources, CPU and number of machines, present in all of the BFT solutions, could pay off if the gain in resilience and availability is essential to the service provided.

The three practical BFT SMR protocols presented above show us that different approaches and optimizations can be taken in order to produce BFT systems. Each of the approaches carries a gain in performance in some cases while sacrificing some of it in other situations. For example, when no faults occur and the network is stable, Zyzzyva has the best performance, due to its fast and two-phase cases. On the other hand, under the presence of faulty clients, Aardvark beats Zyzzyva due to its resilient design. A faulty client can also harm the performance in PBFT, sending inconsistent MAC authenticators, one for the leader and other for the other replicas, this would send the system to recurrent view changes, limiting its progress (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009). Moreover, when comparing PBFT with Zyzzyva, we can denote that the first is more predictable and steady performance under increasing payload sizes of the requests. On the other hand, Zyzzyva offers a better performance in wide-area network, where packet loss is frequent (Singh, Das, Maniatis, Druschel, and Roscoe 2008).

It's also noticeable that, despite their major differences, all the approaches can be decomposed in the same modules. All implementations have a protocol for the agreement and a module for view change. The clients have also different behaviours and responsibilities, but in all of the systems they all request something and act upon a reply (or the absence of it). Moreover, the view carries a slightly different meaning and information with it in the different approaches, but, then again, they are always responsible to capture the configuration and (to some extent) the state of the system in a given timespan. So, a pattern starts to emerge when we look to this protocols from a higher level, which is very important to be able to modularize these in order to adapt them, specially if we are looking for a general solution that can work with arbitrary protocols.

Finally it is important to notice that some protocols, like Zyzzyva, rely on the client to ensure correctness as they actively participate in history consistence and fault detection. This introduces some extra challenges to adaptation techniques which are discussed further in Chapter 3.

2.3 Approaches on Protocol Adaptation

There is a wide spectrum of adaptations that can be made to protocols, that can affect different ranges of replicas, as well as they can demand distinct levels of consistency on coordination. Adaptations can have effect on all replicas or in just a subset, as an example, a change in the agreement protocol must be known by all replicas, while a change in the batch size, usually, only concerns the leader. Moreover, adaptations may demand an atomic agreement (no request can be processed with different settings among the replicas), for example, if the authentication method changes, all replicas must know it before processing any further request, otherwise inconsistencies may be introduce in the system. On the other hand, if it is the time-out threshold to detect replicas failing to answer that changes, most of the times is not paramount that it happens at all the replicas simultaneously to keep the correctness of the system. Below we present some adaptations that can be made for each of the categories.

- **Adaptations that have effect only on a subset of replicas:** Change in the batch size of requests, changes in logging techniques, etc.
- **Adaptations that have effect in all replicas:**
  - **Demanding an atomic agreement:** Changing the underlying agreement protocol; changing the BFT SMR approach; changing the authentication method used for the messages.
  - **Demanding eventual agreement:** Changing the time-out threshold to detect replicas failing to reply;

To carry out such reconfigurations is then needed an orchestration between the replicas of the system, if the adaptation affects more than one replica, and some local reconfiguration strategy (Rosa, Rodrigues, and Lopes 2007). The orchestration component is responsible to coordinate the reconfiguration among the replicas in order to ensure the correct function of the system. This orchestration works much like a synchronization barrier, so that all the replicas move through the reconfiguration steps in sync with each other. This reconfiguration steps are the local reconfiguration strategy, that defines what are the local steps needed to change the configuration.

The combination of local strategies with different kinds of orchestrations can provide several approaches on reconfiguration, each one has different consistency guarantees and different performance (Rosa, Rodrigues, and Lopes 2007):

- **Flash:** In this strategy all the replicas apply the changes locally, without care for other replicas. This is a strategy that introduces little delay during a reconfiguration, although only works for adaptations that do not demand an atomic agreement.
- **Interrupting:** This strategy stops the systems, applies the new configuration and only then starts the system again. This way and atomic agreement for when the new configuration is applied is guaranteed, but the delay introduced is considerably bigger than the previous solution.
- **Non Interrupting:** To try to minimize the delay of an interrupting strategy while maintaining the guarantees of an atomic agreement adaptation, this approach runs both configurations (old and new) simultaneously until the new one is fully functional, then the old can be shut down. Nevertheless, this introduces more complexity in the orchestration than the previous solutions, it also carries the computational overhead of having two configurations running at once.

2.3.1 Adaptation of SMR Protocols

Any adaptation strategy used may ensure the properties of systems that is being adapted. When talking about SMR usually we want to ensure that every request is totally ordered at most once, thus being necessary to ensure that a given request that was already ordered previously to an adaptation is not ordered again.

Another key property of SMR that needs to be ensured during and adaptation is *irrevocability*. This property guarantees that if the systems outputs to the user that some request was executed, then the state of the system must always reflect that execution, even if some processes fail. More informally, if a user successfully executed an action, like depositing some amount of money, then it as assurance that in the future the money will still be in her account, despite what failures can happen, even if the machine where the money was deposited explodes. SMR offers this because otherwise it would be fuzzy to define even the semantics of tolerating a fault. So, when adapting the system we must assure that no request is dropped from the history if a client already knows about its execution.

2.3.2 Configuration as a Dynamic Module

When talking about adaptations in the context of SMR, we can use its own consensus module to execute the orchestration because the synchronization can be made be deciding on the steps to perform. Lamport, Malkhi and Zhou(Lamport, Malkhi, and Zhou 2010) presented a method to produce algorithms for adaptation that uses the inter-replica agreement naturally present in SMR systems to decide on a new configuration, using the already existing interface to propose commands. The agreement is dependent on the configuration, so, to agree about a request i, all the replicas must be using the same configuration. So, to allow the replicas to know how to behave, a specification of the current configuration is kept in the system's state. To change it, there must be an agreement, so a new special request is introduced, RECONFIGURE(C), which specifies the new configuration C. So, when RECONFIGURE(C) is agreed upon as the request i, the configuration is set to C, so from request $i + 1$ and onward this new configuration is used. This method is called by the authors R_1.

Nevertheless, if we aim to adapt the underlying protocol, developing a system like this in practise would have some drawbacks. Firstly, it would be a complex monolithic protocol, due to be a composition of usually already complex SMR protocols, so it would be harder to develop and to prove correct than smaller non-adaptive protocols. On the other hand, it would be harder to extend to keep up with the state of the art, probably becoming obsolete in a short amount of time. An alternative to this is using black-box switching.

2.3.3 Switching Between Black-Box Protocols

We designate by black box-switching the task of building a reconfigurable state-machine from two state-machine implementations that have no support for reconfiguration, not even any special command to put the state-machine in a quiescent state.

In this context we say that a state-machine is reconfigurable if it accepts a special command RECONFIGURE(C, C') that can be applied to configuration C to change the state-machine to configuration C'. Since one aims at providing this abstraction using state-machines that have no support for reconfiguration, the solution consists in instantiating two different state-machines, a state machine $S1$ running configuration C and another state machine $S2$ running configuration C' and, at some point, start redirecting all request to the second state-machine.

There are two main challenges in this approach. The first is how to know that is safe to stop using $S1$ and start using $S2$. The second is to avoid a long hiatus, where no commands are processed, during the switching operation.

A naive look at the first problem could indicate that a simple, yet not efficient, solution to the first problem would be to coordinate all nodes to stop submitting commands to machine $S1$. When one is sure that new commands are no longer being submitted to $S1$, one would simply wait for all commands previously submitted to be ordered, and then one could resume the operation by submitting new commands to $S2$. Unfortunately, this strategy is only feasible in a system with a perfect failure detector. In the general case, it may be hard to ensure that clients and replicas that are not reachable have reached a quiescent state.

Due to the problem above, most solutions rely on using state machine $S1$ to define which is the last command to be ordered by $S1$. This is implemented by issuing a special command that works as a marker. All commands that are ordered after the marker can no longer be processed and need to be resubmitted to state machine $S2$. Note that this approach does not require $S1$ to be made quiescent. $S1$ may still process commands after the marker, but the results are ignored to prevent the duplicate execution of commands, so these need to be re-processed by $S2$.

The approach above solves the first problem but does not address the efficiency problem. In fact, a large number of commands may be affected by the reconfiguration, being ordered after the marked and needed to be re-submitted (to be re-ordered again) to machine $S2$ which may double the latency of command processing during the reconfiguration procedure. In the next paragraphs we discuss some approaches to mitigate this problem.

2.3.3.1 R_α

Lamport et al. presented an improvement to R_1 so that it could deal with parallel commands being decided after a reconfiguration happened. This is, in the R_1 approach, if the agreement $i + 1$ was being agreed upon when a reconfiguration was decided, at command i, $i+1$ would have to be resubmitted to the new configuration, as after the reconfiguration no more decided commands should be processed to ensure that it is not processed by both the old and the new configuration.

In order to allow concurrent processing of requests, the state machine must then delay the change of the reconfiguration when a RECONFIGURE(C) is agreed on. This is, if a RECONFIGURE(C) is decided as the request i, the reconfiguration only takes place after executing the request $i+\alpha-1$, being α the maximum number of concurrent agreements running at a giving time. This allows for the requests that are being agreed upon in parallel with the RECONFIGURE(C) to use the old configuration safely, because it is ensured that a reconfiguration request will only affect processes that are not being decided concurrently. This method is called R_α.

If it is necessary to make the reconfiguration take place immediately after a RECONFIGURE(C), this is, no request will be executed between deciding RECONFIGURE(C) and the configuration taking place, $\alpha-1$ *noop* commands must proposed right after the RECONFIGURE(C). This can be batched in order to consume the same resources as it was only one command.

However, this solution would imply buffering all the commands that arrive during the reconfiguration time, so that they are then processed by the new configuration. This would then cause an interruption on the processing of new requests and a degradation in the quality of service due to a drop in the throughput of operations done.

2.3.3.2 Run-time switching between protocols

Mocito and Rodrigues (Mocito and Rodrigues 2006) try to mitigate the problem of interrupting the processing of new messages during an adaptation event by using an approach similar to the non interrupting approach presented by Rosa et al. (Rosa, Rodrigues, and Lopes 2007). Despite of the work of Mocito et al. being focused on changing between total order protocols, we believe that the ideas present in it can be adapted to SMR, as total order is recurrently a central piece in the development of SMR systems. The main idea of the work of Mocito et al. is to have both protocols, the one that is currently running and its successor, running simultaneously during the switching time (since the adaptation command is issued until the new configuration is fully functional). On a high level view, to switch from protocol A to protocol B, the protocol works as follows:

1. A special message stating the intended switch is broadcast to all processes.
2. When a process receives this message, it starts using both protocols and sends a flagged message notifying this. Although, until the switch is on the final stage, only protocol A messages are delivered. Algorithm B messages are buffered in order.
3. When every process is using protocol B, protocol A is stopped. Then, all messages from B that are buffered and were not delivered by A are delivered in order. Finally, B starts operating as normal.

This approach effectively eliminates the downtime in service that would happen if a protocol was stopped and then the new one was initialized. On the other hand, this solution may introduce a significant overhead on network if the protocols individually operate near the limit of the available bandwidth. This happens because the network, in order to not become a bottleneck, must support both protocols running at the same time. Despite this, in the paper is discussed an optimization to mitigate this issue that consists in sending only the headers of the current protocol (A) during the switch. So, this way, the payload is only transmitted using the new protocol (B), reducing the load on the network.

This solution leaves yet two problems to solve: it does not allow for concurrent reconfigurations and it relies on a perfect failure detector. The first problem is not really a concern to us, as we assume that the adaptation manager present in the Abyss system will not send concurrent reconfiguration commands, as it would be a source of overhead. The second issue is of major importance to us, as we want to develop fault tolerant systems, we can not rely on a perfect failure detector. In case of a replica crashing or getting mute, a switch would never finish because there would be a replica never stating that it was already using the new protocol.

2.3.3.3 Building a Reconfigurable State Machine from Non-Reconfigurable Protocols

Bortnikov et al. (Bortnikov, Chockler, Perelman, Roytman, Shachor, and Shnayderman 2015) further explored this black-box approach, mitigating the problem of having to resubmit some requests to the new configuration, like Mocito et al., but under a crash fault tolerant paradigm. So, the authors of the paper present a framework to develop a configurable state machine from non-configurable ones, assuming only reliable communications (messages sent from a process to another are eventually received) and a crash fault model. The work focuses on the change of the set of replicas participating on the state machine execution at a given point in time, this is, switching between non-reconfigurable SMR implementations that work with a fixed number of replicas. Although, this work can be extended to support the switch between SMR implementations that not only can have different number of replicas, but also can execute in different ways, having distinct patterns of communication, for example.

In this approach, the processes of the reconfigurable state machine must be able to state that they are ready to start processing requests under the new configuration. This is, upon RECONFIGURE(C'), the processes broadcast READY(C') as soon as they are aware of it. When any replica receives READY(C') from a quorum of replicas, then they start using the new configuration and notify all active processes (under all configurations) that they changed its own configuration with a NEW-CONF(C') message.

Switching between different state machines generally carries the overhead of transferring the state from one to another, in order to ensure total order. To mitigate this overhead, this work is built on top of the premise that different configurations (i.e non-reconfigurable state machines) must be independent of each other, so they can start operating from the initial state, independently of the history of the previous configurations.

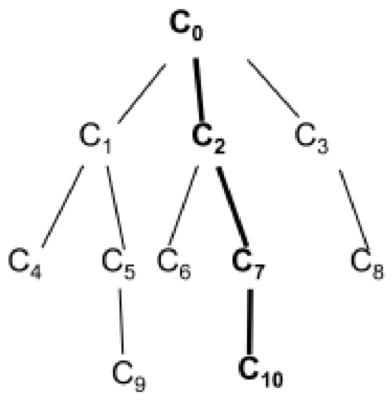


Figure 2.2: A tree of commands and configurations resulting of speculative execution in Bortnikov *et al.* reconfigurable state machine approach. The edges correspond to the history of commands executed between two configurations. The highlighted path corresponds to the history kept in the global history, being the other discarded at some point.

Another optimization introduced, leveraging the independence of the configurations, is the concurrent speculative execution of new configurations, as soon as they are proposed, even if not already decided by the current configuration. This way there is no need for resubmitting requests to a new state machine, as every request that happened during the switching event was already submitted to it. Although, a problem may arise if concurrent reconfiguration changes occur, it would form a tree of commands and configurations (as in Figure 2.2), instead of a single ordering line, thus breaking the total ordering of commands. To solve this, all replicas prune this tree in a deterministic way, by choosing always the first reconfiguration decided to be the one that is kept in the global ordering as the next. To inform this decision to the configurations that are being executed speculatively, the state of each configuration is shared periodically among the configurations, so they can prune the tree accordingly.

Although the dealing with concurrent reconfiguration requests is not a concern for our work, this speculative approach reduces the delays of deciding and starting a new configurations, specially under high-latency networks, mitigating also the need to buffer or resubmit commands that were being processed during the reconfigurations. To ensure that there are no commands duplicated by executing speculatively, the reconfigurable state machine only proposes new commands to the currently operating state-machine configuration if no reconfiguration was already decided by that configuration. If some reconfiguration was already decided, then, by design, some other state-machine implementation is already responsible for ordering such commands,

To use this SMR protocol in practise there is a need an additional component, the Command Queue (CM). This component is responsible for associating the clients' commands with configurations, proposing it in the running configuration or configurations, if concurrent speculative configurations are executing. So, this component must be aware of the state of the active configurations in the system, which does by keeping track of the *ready* messages, to know which configurations are running at the time.

2.3.4 Stoppable Protocols

Despite the black-box approach being already proven as a feasible solution, it would be easier if the different state machines had a *stop* primitive to put it in a quiescent state. This way we can be sure that at some point in time a given configuration has stopped, having a giving final state, and it would not process further requests. This would allow for a simpler management of the adaptable system, even under a fault-tolerant paradigm, as by design, it would be possible to know when a machine has stopped and when it would be safe to send new requests to other state machine.

The basic idea behind this stopping kind of adaptation is to send a stop-sign to a given state machine, and then the state machine stops executing requests. All the requests addressed to the machine will receive an answer stating that the machine has stopped, and eventually some kind of pointer to the new one (Lamport, Malkhi, and Zhou 2010). Once again, if multiple agreements are run in parallel further care must be taken. The stop-sign would impair sending an execution guarantee to the client as soon as the request is agreed upon, it would have to wait for all previous requests to be executed, as the machine could stop before. In this case, like the $R\alpha$, a delayed stop-sign must be used, so the stop happens after α agreement instances, guaranteeing that no execution guarantee is violated.

In the same paper is also discussed another way to stop a machine, sending infinite noop requests. This derives from the conceptual thinking of a machine execution consisting in some finite non-noop requests followed by infinite noops. This can be easily represented in finite manner, while using batching.

2.3.4.1 The Next 700 BFT Protocols

These ideas of stopping the running replicas and spawning new ones with a different configuration were explored with a BFT mindset in Abstract (Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015). The main idea is the development of several components, called Abstract instances, that, at a given time, one instance is running and providing service to the client, when some event of interest happens, an adaptation event occurs and other Abstract instance takes its place. This adaptations event is a generalizations of a *stop-sign*, as it is a deterministic condition checked by each replica themselves, it can be a *stop-sign* sent by another process (e.g. the client) or any execution or environmental condition, as some time-out or deciding a given amount of commands to be executed. Moreover, this solution deviates from other BFT SMR, like PBFT and Zyzzyva, as it may abort client's requests.

More concretely, an Abstract instance implements a BFT protocol specialized for the given system conditions, called progress conditions, as it only needs to guarantee progress under that conditions. If progress conditions are not met (i.e some assumed condition fails), the Abstract instance aborts, and other with weaker system assumptions takes its place. Therefore, to guarantee the correct functioning of the whole system, it is necessary to guarantee that no request is aborted by all instances. As an example, it is possible to have an Abstract instance that only makes progress when there are no faults, aborting otherwise, in this case it is necessary to ensure that some other instance is capable of dealing with faults, or a client would never get its request answered. As suggested by the authors, this is usually achieved by using a robust well-studied BFT protocol as one of the Abstract instances.

The transition between instances is mediated by the clients, as they propagate the history of an aborted Abstract instance to the next one. So clients need to be aware of the adaptations happening in the server replicas, not only for interacting with them correctly, but also to carry the state. The transition happens in three steps: stopping the current instance, choose a new one and finally initializing the chosen new one. As choosing what adaptation to do is out of the scope of this work, so next we will describe in more detail only how an instance is stopped and initialized:

1. **Stopping the current Abstract instance:** An Abstract instance stops when it aborts the first client request, due to the violation of the progress conditions. Along with the abort notification, an *abort history* is also sent. The mentioned history contains, as prefix, the *commit history* of the instance, and possibly some uncommitted requests.
2. **Initializing the new Abstract instance:** The client invokes the new Abstract instance with the *abort history* of the previous instance. This history is used to define the initial state of the instance, before it starts processing new requests. Abstract does not need any explicit agreement to decide the one common *abort history* among the replicas of an aborting instance. This is possible because , by design, every *abort history* as the same *commit history* as prefix, being this enough to make possible the guarantee of total order.

If switching through a client is a problem, in the context of some specific system, it is possible to extend Abstract to allow switching through a component responsible for the reconfiguration, or even to do the switch through the replicas. In the first case, an aborting Abstract instance must send the abort notification and the *abort history* to the dedicated reconfiguration component, and this component is responsible to invoke the new instance with the history received. The latter case, switching through replicas, is possible by making each replica act as a client, being able to send a *noop* command, it would mediate the instance switching without making any modification on the state of the system.

Having this switching mechanism through the client, without any pre-emptive action to start new Abstract instances could carry the weight of making a client wait for the new instance to initialize, as opposed to the solution of Bortnikov et al.

Abstract presents itself as powerful solution to facilitate the development of adaptive BFT systems. It allows for better system qualities (e.g. throughput, latency) by allowing the use of specialized protocols for each situation. On the other hand, it alleviates the difficulty of developing BFT systems, by making possible the development of several simpler modules, that as a whole implement a full BFT system.

2.3.5 Further Considerations

The problem of adapting protocols in run-time is recurrent under different contexts. In this section we will explore some interesting ideas on adaptations on systems that were not developed to adapt SMR systems, but have some interesting considerations about adapting distributed systems in general, which can be used in the BFT SMR context.

2.3.5.1 Multi-step Algorithm Switching

Chen, Hiltunen, and Schlischting explored these ideas of non-stopping adaptation, and developed an architecture and method to build a distributed system that is able to adapt between configurations gracefully (Chen, Hiltunen, and Schlichting 2001). In this approach, every adaptable component (e.g. total order component) has several *adaptation-aware algorithm modules* (AAM) and a *component adaptor module* (CAM). The first type of module is responsible to provide some specific implementation for the module functionality. The CAM is responsible for managing the adaptation of the module, this includes switching safely between distinct AAMs, coordinating the change with the other replicas in the system and detecting when and which adaptation should occur.

Generally, switching between protocols in a distributed system must take into account the messages flowing in the network, as they may end up being received by a replica operating on a new protocol that does not recognize it. The authors of the paper try to solve this in a seamless manner by using a three-step change algorithm. Upon a adaptation event, the component prepares to receive messages from the new AAM, as well as control messages for the switch-over process. When all components across all replicas are prepared, they start processing the outgoing messages with the new AAM. Finally, when all replicas are sending messages with the new AAM, the component can stop receiving messages from the old AAM. The AAM must be aware of the adaptation process, as it must provide an API to execute the three steps described earlier. As both modules are active during the switch-over, incoming messages must be identifiable as being from the old AAM or the new, this is easily achievable by using specific headers or a similar mechanism.

This approach presents itself has a modular and understandable approach for reconfiguring a distributed system seamlessly, without stopping the whole system in order to carry out change, being more general than Bortnikov's *et al.* approach and introducing less load on the network than the solution of Mocito *et al.*. Nevertheless, it demands the protocols used in adaptable components to have awareness of the adaptation process. This way, the developer must adapt the protocols, what can be a drawback, because understanding some protocols implementation can be a hard and time-consuming task.

2.3.5.2 General *vs* Specialized Switchers

Couceiro explored the use of a general stopping approach *versus* the use of specialized, custom built, non-stopping switching mechanisms (Couceiro, Ruivo, Romano, and Rodrigues 2015). In this work, the authors implemented a framework to allow the adaptation of protocols in the context of replicated in-memory transactional systems. This framework allowed to have different reconfiguration protocols to switch between protocols, having a general ”stop and go” approach, as well as being able to support custom tailored mechanisms to switch between two particular protocols.

The ”stop and go” approach described by the authors demanded a given protocol for replicated transactional systems to have a *boot()* primitive, that initializes the said protocol from inactivity, and a *stop()* primitive that stops the protocol, putting it in a quiescent state. On the other hand, the specialized switching algorithms described take advantage of the particularities of the protocols. The solution presented used a similar approach to the one of Mocito et al. (Mocito and Rodrigues 2006), with both protocols running at the same time during the transition. Although, it deviates from it because how and when each protocol really processes requests is decided by the switching algorithm itself, so, they may be running in parallel, but only one processing requests, for example.

A practical comparison between both approaches is presented in the paper, and, as expected, the drop in performance when executing a custom switch is much smaller than when compared with the general approach. This happens because the specific approach leverages the white-box approach of the algorithms to perform a switch without having to interrupt the processing of new requests, or with very little interruption. The big downside of these specialized approaches is that they demand for a deep understanding of the specifics of the protocols to be switched. Moreover, switching between some protocols can be a very hard problem to solve, if not impossible, due to incompatibilities that often occur between different approaches.

2.3.6 Discussion

Building adaptive SMR systems switching among other (static) SMR implementations as black-boxes seems a more feasible approach then developing a monolithic, complex and hard to manage and extend intrinsically adaptable SMR protocol. Nevertheless, this is done ate the cost of building an adaptation coordination capable of performing the switch among two different state machines efficiently while maintaining the correctness of the overall system. To ease this coordination one could have the different state machines to be stoppable, this is, having a stopping primitive to put them in a quiescent state, where it is guaranteed to make no further progress. Nevertheless, this implies the development of such kind of protocols, either from scratch or adapting already existent ones, which can be a complex and costly task which a practitioner may not want to endure. As an example, if we analyse the differences between Paxos (Lamport et al. 2001) and a stoppable version of it (Lamport, Malkhi, and Zhou 2008), we can see that deriving stoppable protocols may be not as trivial as it seems upon a first look.

When talking about adaptation of the underlying protocol in a SMR systems, the client usually needs to be somehow aware of the adaptations that are happening in the server replicas, because, as discussed earlier, the client can play different roles in different protocols, specially in BFT SMR. Messages flowing in the network may also have incompatible formats, which can introduce further challenges. As an example, if we use a black-box specific SMR implementation as a configurable parameter of our adaptive the state-machine, switching between a protocol that uses symmetric cryptography to another that uses signatures is not trivial because messages already sent by clients in the older protocol can not be processed in the new one. Given this, either the client gets its request refused and must resend in a new message, much like the stop-sign approach, or the message must be translated, which can be hard as the private key of the client is secret. Therefore, we can denote that both non-stopping and stopping approaches get very similar in this case. So, we can denote that the optimizations introduced by concurrently or speculatively ordering new commands under a new configuration may not be useful to its full extent, as the client must resend its request with a new format anyway.

In summary, both approaches, black-box and using stoppable protocols, are different paths to achieve a goal but, due to optimizations and implementational details they can produce similar algorithms, as stated in (Lamport, Malkhi, and Zhou 2010). So, discussing if a non-stopping approach performs better than a stopping approach is complicated because there is no comprehensive study comparing the two. Although we think that the more specialized approach we take when doing a reconfiguration, the least overhead will be introduced (Couceiro, Ruivo, Romano, and Rodrigues 2015). Therefore, in this work we will try to implement several approaches, more general and more specialized, in order to compare them in practise.

Deciding which configuration to use and when to do it is also a concern when talking about adaptive systems. However, that is out of the scope of this work, as it is part of the Abyss project, where these decisions will be made by other components.

Summary

We have started by introducing and comparing some of the most relevant existing BFT protocols. It stood out that no protocol outperforms the others for every case, as they are optimized for some case assumed to be more frequent. For example, Zyzzyva performs better when no faults occur whilst PBFT beats Zyzzyva when faults do happen. Because the environment of execution is very dynamic and changes over time, it is relevant to dynamically reconfigure a system so the optimal solution for the current environment is used.

Given the need to reconfigure the system dynamically, we also surveyed existing work on dynamic adaptation of protocols. Most of the previous work in this area assumes a CFT system model. Furthermore, for the few works that take Byzantine faults into account, to the best of our knowledge, there is no available comparison of their performance. This leaves a systems designer who wants to develop an adaptive system without data to make informed decisions about which solution to choose.

To address this gap, in the next chapters we present a conceptual framework to reason about reconfiguration strategies, a portfolio of these strategies (that adapt previous techniques designed for other fault models), and an experimental comparison of the performance of these techniques in practice.

3 Byzantine Adaptation

This chapter starts by presenting a common conceptual framework to reason about the different adaptation techniques. This conceptual framework was designed to allow an easier comparison and understanding of the different approaches on adaptation that appear in literature, which are described for different use cases and using distinct names for similar core concepts. The different strategies are therefore divided in three main categories: i) *adaptation of reconfigurable protocols*, ii) *adaptation of protocols that are modelled as black boxes*, and iii)*adaptation of stoppable protocols* matching the categories described in Chapter 2.

Then, it discusses how these techniques can be applied in a SMR BFT context. On on side, there are some particularities of the operation mode of most SMR implementations, specially due to the client/server operation. On the other side, some modifications must be made to the adaptation algorithms, so they can cope with Byzantine faults. Some optimizations are also presented and discussed.

Lastly, the chapters presents the design and implementation of the a number of techniques for the BFT-SMaRt framework. A general architecture, capable of using distinct SMR implementation as black-box processes is also discussed at the end of this chapter.

3.1 System Model

This work assumes a Byzantine fault model where processes can present an arbitrary behaviour, including colluding to attempt to harm the system with a coordinated attack. Nevertheless, the adversary is considered to have limited computational power and cannot break the cryptographic primitives used in the algorithms. It is assumed a partially synchronous network, where arbitrary periods of asynchrony may happen, but eventually there is a period of synchrony where the system can make progress. In this periods of synchrony, the transmitted messages shall arrive at their destination within a known time bound.

The system operates with N replicas of computation with the ability to instantiate several Byzantine consensus protocols. This set of replicas is able to tolerate a defined number of faults, being henceforth referred as f. Moreover, the clients of the system do not interact directly with the replicas, but with a *switcher*. This is a module responsible to retransmit clients' requests to one or more consensus protocols, making the adaptation transparent to the client. It is assumed, also, that there is an external module, called *adaptation manager*, responsible to issue the adaptations to the system. When it is necessary to adapt the system, e.g. change the consensus protocol, the adaptation manager sends a command stating that to all the system replicas. This module may also be replicated to tolerate Byzantine or crash faults, so the described system is designed to support such replication, only starting an adaptation if some quorum of adaptation manager's replicas issued the same adaptation.

The system is designed considering only consensus protocols with no tentative or speculative executions. This is, no correct replica may have an inconsistent state when compared other correct replica, where an inconsistent state is having a different operation ordered in the same logic time frame. The commutation between such protocols would imply the development of a specific switching mechanism that was able to deal with the particularities of the behaviour of such protocols. Furthermore, the client must not intervene in the protocol correctness. This would imply that the client was aware of the adaptations in the system and demand a much more complex switching algorithm to adapt also the client behaviour. Besides that, a liveness problem could arise if a client is always behind the current system's configuration, getting its requests denied indefinitely until a period of synchrony happens in the network. Even so, there would be the need to deliberately restrict the frequency of adaptations, crippling the adaptive performance of the system. Zyzzyva is an example of both cases, it uses speculative executions and relies on clients for correctness.

3.2 Byzantine Fault Tolerant Adaptation Techniques

3.2.1 Adaptation of Reconfigurable Protocols

The adaptation using reconfigurable protocols, described by Lamport, Malkhi and Zhou (2010), is general an can be applied in a Byzantine context without significant changes. The correction of the technique depends only of the properties of the underlying protocol, so if a BFT protocol is used, its reconfiguration also supports this kind of faults.

Nevertheless, as this systems aim at supporting a replicated adaptation manager, it is necessary to develop an additional mechanism to do so. This mechanism consists in collecting enough (usually $f+1$) similar, properly signed, adaptation requests from the manager's replicas. Furthermore, when dealing with Byzantine faults, it is also necessary to prevent that a replica can submit an adaptation command if the said operation was not requested by the manager, as it can lead to an attack that degrades the system's performance by inducing unwanted adaptations. We considered two approaches to mitigate this problem. The first one is to include a cryptographic proof of the rightfulness of the adaptation command. This proof includes the signature of a quorum of the adaptation manager's replicas. The second approach is to only start an adaptation after receiving at least $f + 1$ commands, where f is the number of tolerated faults, before starting an adaptation. This approach would not need to use cryptographic proofs because with $f+1$ commands at least one would be correct. Nevertheless it would delay the introduction of the adaptation, as this last solution needs $f + 1$ consensus runs, instead of just one. For this reason the first solution was implemented in the developed system and a representation

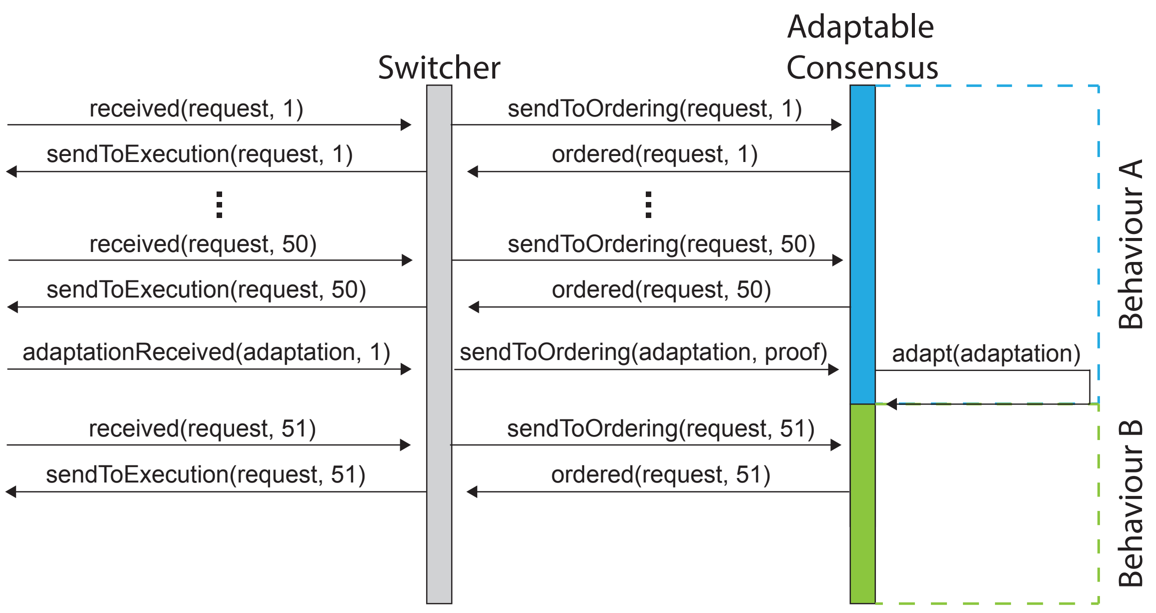

Switcher
Adaptable Consensus
received(request, 1)
sendToOrdering(request, 1)
sendToExecution(request, 1)
ordered(request, 1)
received(request, 50)
sendToOrdering(request, 50)
sendToExecution(request, 50)
ordered(request, 50)
adaptationReceived(adaptation, 1)
sendToOrdering(adaptation, proof)
adapt(adaptation)
received(request, 51)
sendToOrdering(request, 51)
sendToExecution(request, 51)
ordered(request, 51)
Behaviour A
Behaviour B

Figure 3.1: Flow of an adaptation using reconfigurable protocols

of the algorithm is presented in Algorithm 1. Note that this mechanism is orthogonal to all the described strategies. A sketch of the overall flow of an adaptation is presented in Figure 3.1.

This is expected to be the most efficient approach to perform adaptations in runtime, nevertheless developing such protocols may prove itself to be harder than using already described protocols. Furthermore, there is the chance to make the protocol prioritize reconfiguration commands. This is possible because this technique is aware and can manipulate the actual queue of incoming requests and can always put a reconfiguration request ot the head of the queue, instead of putting it on the tail, like an ordinary request.

In the context of this thesis, the problem of concurrent consensus is not an issue as BFT-SMaRt uses batching instead of concurrent consensus as a performance optimization. In its turn batching presents no special challenge when applying an adaptation, it's only needed that the ordered batch is identified as having an adaptation so it is applied upon ordering.

Using adaptable protocols also makes possible the introduction of finer-grained adaptations, like changing the leader, without having to swap the entire protocol and deal with pending requests transfer between protocols (like with other type of techniques, described further on this chapter). Some of this adaptations may demand a consensus run to decide the right logical moment when to apply the adaptation across all replicas, so it is applied logically at the same time to ensure correctness. On the other hand, some adaptations, like changing the batch size or the fault detection time-out, without running a consensus, similar to the *flash* adaptations discussed in (Rosa, Rodrigues, and Lopes 2007). This changes don't affect correctness even if they are not applied at the same time in all replicas and, if no consensus is executed, the adaptation is usually applied faster.

The development of an adaptable protocol in BFT-SMaRt was facilitated because the Mod-SMaRt framework (Sousa and Bessani 2012) already has an isolated consensus module and some configuration parameters by design. This way, a second consensus protocol was implemented and the consensus module was generalized as a configurable parameter. This is discussed in more detail in section 3.3.

Algorithm 1 Reception and Execution of Adaptation Commands

```
1: procedure onAdaptationReceived(adaptation, sender, signature)
2:     if sender is trusted $\wedge$ signature is valid then
3:         PendingAdaptations $\leftarrow$ PendingAdaptations $\cup$ (adaptation, sender, signature)
4:         checkPending($adaptation$)
5: procedure checkPending(adaptation)
6:     similarAdaptations $\leftarrow$ similarAdaptations(PendingAdaptation, adaptation)
7:     if length(similarAdaptations $\geq$ quorum then
8:         if adaptation needs ordering then
9:             proof $\leftarrow$ makeProof(similarAdaptations)
10:            sendToOrdering(adaptation, proof)
11:        else
12:            executeAdaptation(adaptation)
13: function simliarAdaptations(adaptation)
14:     similarAdaptationCommands $\leftarrow \emptyset$
15:     for all pending in PendingAdaptations do
16:         if pending.adaptation $\equiv$ adaptation then
17:             similarAdaptationCommands $\leftarrow$ similarAdaptationCommands $\cup$ pending
18:     return similarAdaptationCommands
19: procedure onOrdered(adaptation, proof)
20:     if (adaptation, proof) is sound then
21:         executeAdaptation(adaptation)
```

3.2.2 Adaptation of Protocols Modelled as a Black-box

To adapt an non-adaptable, “black-box”, protocol, it is necessary to send a marker (a special control message or a flag associated with an ordinary message) in the active protocol (A) to decide the logical instant, in the message flow, where the switch to the next protocol (B) takes place (Mocito and Rodrigues 2006). To facilitate the implementation, this marker may contain the information needed about the new configuration, so every replica is able to apply it correctly without further messages from other modules. This way, the marker acts like an adaptation command but which execution is performed at the switcher, instead of the protocol itself. A sketch of the flow of an adaptation using protocols as black-boxes is presented in Figure 3.2.

Nevertheless, in leader-based protocols, like most BFT consensus protocols, there is an additional challenge. It arises from the following: if the process p_i is the leader of protocol B but it’s the last process to be informed, due to, for example, network asynchrony, that A ordered a marker, it may be suspected as faulty despite being correct. This happens because all the other processes, being already operating in B, may notice the lack of activity of the leader and begin a leader change, even before all processes are participating in B. In most cases it does not break the protocol’s correction, as this lack of activity is similar to a period of asynchrony in the network, falling in the system model of most of the recent BFT consensus protocols, however it may impair the performance and progress of the system.

For this reason, in a Byzantine fault tolerant system, it may be preferable that a switcher starts re-

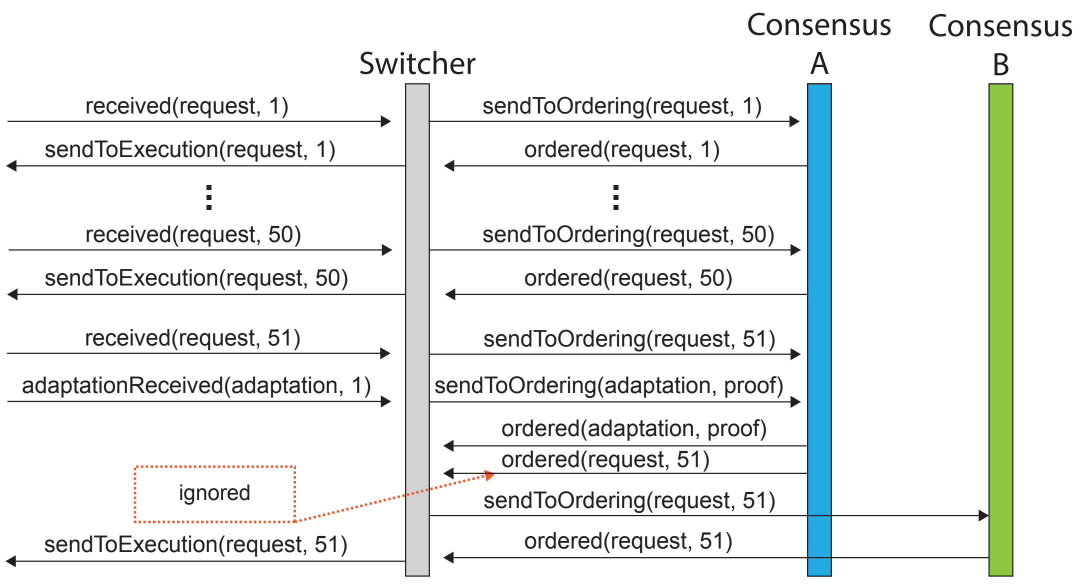

Switcher
Consensus A
Consensus B
received(request, 1)
sendToOrdering(request, 1)
sendToExecution(request, 1)
ordered(request, 1)
received(request, 50)
sendToOrdering(request, 50)
sendToExecution(request, 50)
ordered(request, 50)
received(request, 51)
sendToOrdering(request, 51)
adaptationReceived(adaptation, 1)
sendToOrdering(adaptation, proof)
ordered(adaptation, proof)
ordered(request, 51)
ignored
sendToOrdering(request, 51)
sendToExecution(request, 51)
ordered(request, 51)

Figure 3.2: Flow of an adaptation using protocols as a black-box

transmitting requests to B as soon as possible, even before A orders a marker. This causes a possible parallel execution of both protocols and further measures must be taken to ensure safety (Mocito and Rodrigues 2006). Messages ordered in B should be frozen until A becomes inactive, then a sanitization process shall be done to prevent duplicated messages being delivered to the application. This process consists in removing all the frozen messages that were already delivered by A. Finally all retained messages from B can be delivered and it may proceed delivering any further ordered messages.

For simplicity, a version of the algorithm which does not use parallel execution is described in Algorithm 2. The parallel execution of protocols is discussed in more detail in Section 3.2.4.

Algorithm 2 Execution of Adaptations using Stoppable and Non Reconfigurable Algorithms

```
1:  procedure RECEIVEREQUEST(ordinaryRequest)
2:      pendingRequests ← pendingRequests ∪ ordinaryRequest
3:      SENDTOORDERING(ordinaryRequest)          ▷ Order in the currentAlg algorithm
4:  procedure ONORDERED(ordinaryRequest, algorithm)
5:      if algorithm is currentAlg then
6:          pendingRequests ← pendingRequests \ ordinaryRequest
7:          SENDTOEXECUTION(ordinaryRequest)
8:  procedure EXECUTEADAPTATION(adaptation)
9:      currentAlg ← adaptation.algorithm
10:     SENDTOORDERING(pendingRequests)
```

In addition, note that, in the case of SMR, where the operations come from clients asynchronously and in parallel, the ordering of requests in the queue awaiting for ordering is even more unpredictable than in cases where the operations are issued by the replicas themselves. This has the potential to make the number of requests in the queue after a marker to be of considerable size. So, it is likely that some resources are wasted ordering operations after a marker, as they are discarded and re-submitted in the next protocol, even if every switcher stops sending new commands to an protocol after sending a marker to it. Even more so, such action may induce sub-optimal behaviour of the protocol after the marker is ordered, as queue-sharing and leader-election sub-protocols may take place. This is, as the leader may be deprived of some client requests, because they were not submitted by the switcher after a marker, other replicas that have the said requests may retransmit them or even suspect a faulty leader.

3.2.3 Adaptation of Stoppable Protocols

As discussed in Chapter 2, a possible way to mitigate the waste of resources, caused by executing consensus to order requests in an protocol that is no longer active, is to use stoppable protocols. This kind of protocols don't perform any consensus after ceasing to be active, replying asynchronously with the information that it has stopped for every request submitted.

In a CFT system a straightforward way to implement the switch between two stoppable protocols, A and B, consists in executing the following two-step algorithm:

1. Each switcher sends a stop command to A and stops retransmitting new requests until it acknowledges that A is in a quiescent state.
2. The switcher proceeds to deliver all the retained requests, as well as any new ones, to protocol B.

In practise, with our architecture, with a switcher that is omniscient of the client requests and the ordering of them by all protocols, using stoppable protocols is very similar to using them as black boxes, just with the particularity of the marker acting as a stop command when ordered.

3.2.4 Using Parallelization in Adaptation

As seen before, when stoppable protocols or protocols as black-boxes are used to carry out adaptations, a hiatus may occur between ceasing the activity the currently active protocol and start executing operations in the new one. The solutions described in (Bortnikov, Chockler, Perelman, Roytman, Shachor, and Shnayderman 2015) and (Mocito and Rodrigues 2006) minimize this overhead by transmitting all messages in both protocols, during the adaptation time. They assume as well that both protocols are already activated and no process uses any of the protocols incorrectly, this is, does not start operations in an protocol that is not active or a candidate to be as so after an adaptation. The algorithm is described in Algorithm 3.

However, in a Byzantine context, there is a chance that a Byzantine process starts sending messages to any protocol, even if it does not comply with an issued adaptation. This may waste resources and make possible a malicious vector of attack to cause starvation and reduce the system's performance. We found two ways to minimize this problem, starting the protocols only when they are needed to perform an adaptation or sending a proof of the rightfulness to use a protocol because there was a adaptation request that commanded so.

In the first approach, the switcher only starts a protocol if it is necessary to perform an adaptation, and promptly deactivate any protocol that ceases to be necessary in the current execution context. This way no Byzantine process can harm the system's performance by operating incorrectly in some inactive protocol because other processes are not listening to that messages. However, this solution, by starting protocols in run time, may increase the time needed to perform an adaptation as it has to wait for the establishment of communication channels, possible state transfers and leader elections, which are common in BFT protocols.

The second approach assumes all protocols are already running, although some are idle due to being inactive, mitigating the time needed for their initialization during an adaptation. So, any protocol which starts operating in an idle protocol sends a proof, along with the first message sent in the protocol, of the correctness of this action. This proof consists, similar to what was discussed earlier, in a cryptographic proof with the signature of a quorum of adaptation issuer replicas. This quorum may be $f+1$ if up to f faults are tolerated and it is assumed that no correct adaptation issuer instance can send an incorrect adaptation request. When some process receives the first message in an idle protocol, it checks the proof and, if it's considered valid, then the protocol is marked as operating. If the proof is forged, then an accusation about the sending process is issued against the replica that used it and so it may be considered faulty and removed from the working set of replicas.

3.2.5 State Management During an Adaptation

The switcher is responsible to control which ordered messages are sent to execution and added to the history of the system. This happens either directly, managing the queue of requests to send to execution, when parallelization or non-adaptive protocols are used, or indirectly by sending requests to a given protocol only when no other is operating, when using stoppable protocols.

The consistency of the history across all configurations can be achieved in several manners. One of them is to build the whole history as a composition of ordered stable trunks. Each trunk has a common agreed final state of the execution of a protocol and trunks are ordered following the succession of active protocols. This may imply running a state-transfer protocol when adaptation takes place, which introduces a penalty in throughput and delay of its application.

To mitigate this drawback, it is possible to eliminate this state transfer, if the protocols don't have speculative ordering and the adaptation is totally ordered in the history. With these two assumptions, if any replica receives a reconfiguration command, then it already has all the history up to that point. Any deviation in state is managed by the internal state transfer mechanisms of the current protocol.

Furthermore, the queue of unanswered requests must also be taken care of, so no request is lost or duplicated, causing faulty behaviour. For example, losing unanswered requests through an adaptation may impair the detection of a faulty leader which is omitting some of the clients' requests. One way is for the distributed switcher to agree on a common set of unprocessed requests to deliver to the new

Algorithm 3 Reception and Execution of Adaptation Commands Using Parallelization

```
 1: procedure onAdaptationReceived(adaptation, sender, signature)
 2:     if sender is trusted $\wedge$ signature is valid then
 3:         PendingAdaptations $\leftarrow$ PendingAdaptations $\cup$ (adaptation, sender, signature)
 4:         checkPending($adaptation$)
 5: procedure checkPending(adaptation)
 6:     similarAdaptations $\leftarrow$ similarAdaptations(PendingAdaptation, adaptation)
 7:     if length(similarAdaptations $\geq$ quorum then
 8:         proof $\leftarrow$ makeProof(similarAdaptations)
 9:         sendToOrdering(adaptation, proof)
10:         initParallelAlg(adaptation)
11: procedure initParallelAlg(adaptation)
12:     FrozenOrdered $\leftarrow \emptyset$
13:     onSwitchProcess $\leftarrow true$
14:     nextAlg $\leftarrow$ adaptation.protocol
15: function simliarAdaptations(adaptation)
16:     similarAdaptationCommands $\leftarrow \emptyset$
17:     for all pending in PendingAdaptations do
18:         if pending.adaptation $\equiv$ adaptation then
19:             similarAdaptationCommands $\leftarrow$ similarAdaptationCommands $\cup$ pending
20:     return similarAdaptationCommands
21: procedure onOrdered(adaptation, proof)
22:     if (adaptation, proof) is sound then
23:         executeAdaptation(adaptation)
24: procedure executeAdaptation(adaptation)
25:     onSwitchProcess $\leftarrow false$
26:     notSentFrozenOrdered $\leftarrow$ FrozenOrdered $\setminus$ orderedRequests
27:     currentAlg $\leftarrow$ nextAlg
28:     sendToExecution(notSentFrozenRequests)
29:     notSentFrozenOrdered $\leftarrow \emptyset$
30: procedure onReceived(ordinaryRequest)
31:     sendToOrdering(currentAlg, ordinaryRequest)
32:     if onSwicthProcess then
33:         SendToOrdering(nextAlg, ordinaryRequest)
34: procedure onOrdered(ordinaryRequest, protocol)
35:     if onSwitchProcess $\wedge$ protocol is nextProtocol then
36:         FrozenOrdered $\leftarrow$ FrozenOrdered $\cup$ ordinaryrequest
37:     else if protocol is currentProtocol then
38:         orderedRequests $\leftarrow$ orderedRequests $\cup$ ordinaryRequest
39:         sendToExecution(ordinaryRequest)
```

protocol after an adaptation. Nevertheless, in the context of the described solution for BFT-SMaRt this is not needed, as the fault-detection and queue-sharing mechanisms are shared among the different protocols.

3.3 Implementing Adaptation Within BFT-SMaRt

3.3.1 BFT-SMaRt

There are available several open-source libraries to ease the development of BFT SMR systems, namely UpRight (Clement, Kapritsos, Lee, Wang, Alvisi, Dahlin, and Riche 2009), Archistar (Lorünser, Happe, and Slamanig 2014) and BFT-SMaRt(Bessani, Sousa, and Alchieri 2014). The first was the first attempt, that we are aware of, of a library concerned about simplifying the development of BFT systems. However, these concerns introduced a significant overhead in the performance of the system (Bessani, Sousa, and Alchieri 2014). Moreover, as of the date of the writing of this work, the library seems to not be maintained any more, being the last release dated of the 27 of January of 2010. Archistar is a compact BFT replication engine, although it is not easily extensible, because it is a monolithic approach. As for UpRight, it is not currently maintained, and, moreover, has very little documentation, which can make implementation problems harder to solve, in the future.

Given this, in this section we will present BFT-SMaRt, which will be the starting point of our work, because it is still maintained and it is highly modular, facilitating its extension. Besides this, some members involved in the Abyss project have deep knowledge about it, making it easier to solve implementation challenges, when they arose.

This library, implemented in Java, implements a BFT system similar to PBFT. Although it deviates from PBFT, as it uses a modular approach, instead of a monolithic one, to develop something more understandable and extensible. To develop an application using BFT smart, a developer needs only to implement the usual *invoke(command)* on the client, and an *execute(command)* on the server side, leaving all the responsibilities to ensure BFT to the library itself. Besides this, if more complex behaviour is needed, BFT-SMaRt can be extended using plugins, alternative calls and call-backs.

BFT-SMaRt is also able to add or remove replicas form a given system, carrying out the state-transfer needed to initialize the new replicas. This state capture and transfer is isolated in a layer between the replication protocol and the application, so it does not influence the consensus protocol. To carry out such tasks three principles are used: logging the operations executed in the system, taking snapshots of the progression of the system, in different points in time at different replicas (so the system does not stop) and, finally, transferring the state to fresh replicas in a collaborative fashion, with distinct replicas sending different chunks of the state to the initializing replica.

The architecture of BFT-SMaRt, depicted in Fig. 3.3, has the following main modules:

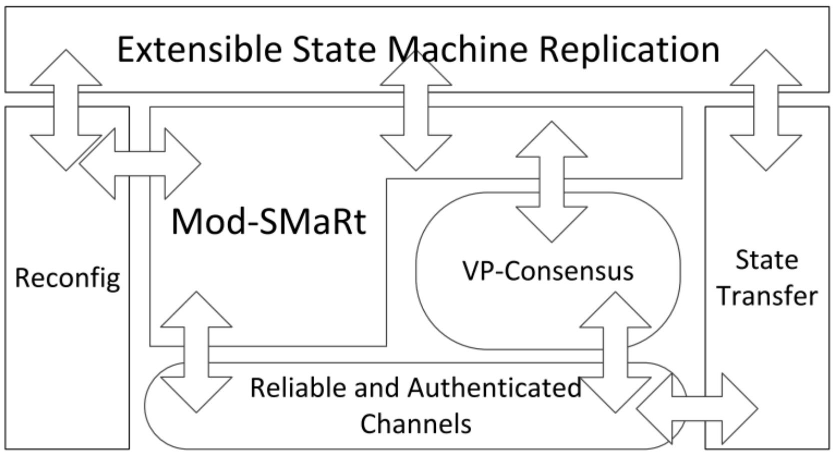

Extensible State Machine Replication
Mod-SMaRt
Reconfig
VP-Consensus
State Transfer
Reliable and Authenticated Channels

Figure 3.3: The architecture of BFT-SMaRt

- Extensible State Machine Replication: responsible for implementing the application behaviour.
- Mod-SMaRt and VP-Consensus (Sousa and Bessani 2012): responsible for implementing the SMR mechanics, including total ordering.
- Reconfiguration Module: responsible for carrying out the addition or the removal of replicas.
- State Transfer Module: responsible for initializing new replicas with the current state, or even recovering the whole system.

BFT-SMaRt was used in a work already done in the area of BFT protocol adaptation (Sabino, Porto, and Rodrigues 2016). In this work, Sabino, developed an architecture, based on BFT-SMaRt, to develop adaptable BFT SMR systems, including a component to monitor the environment of the system and an adaptation manager, that reacts upon the happening of certain events, like a change in the network conditions. Despite this, the adaptations explored in this work are changes in protocol parameters like the batch size and the number of replicas, not discussing the main concern of our work, protocol commutation.

3.3.2 Architecture

To implement the various techniques in order to experiment and compare them, several modifications and extensions were made to BFT-SMaRt. Firstly, since BFT-SMaRt had only one consensus protocol available, an adaptation of the protocol described in (Guerraoui and Rodrigues 2006) to the Mod-SMaRt framework, a partial version of the “Fast Byzantine Consensus” described in (Martin and Alvisi 2006) was developed. This new protocol is called Fast-SMaRt. In addition, to experiment how the adaptation techniques can cope with heavy load in the system, a version of Mod-SMaRt capable to deal with some vectors of performance-degradation attacks was developed. This variation of Mod-SMaRt is called Safe-SMaRt. Moreover, a switcher module was added, responsible for managing the communication between the clients and the existent protocols and configurations. This switcher is ready to receive adaptation command from a replicated adaptation issuer, which is being developed by other members of the Abyss project.

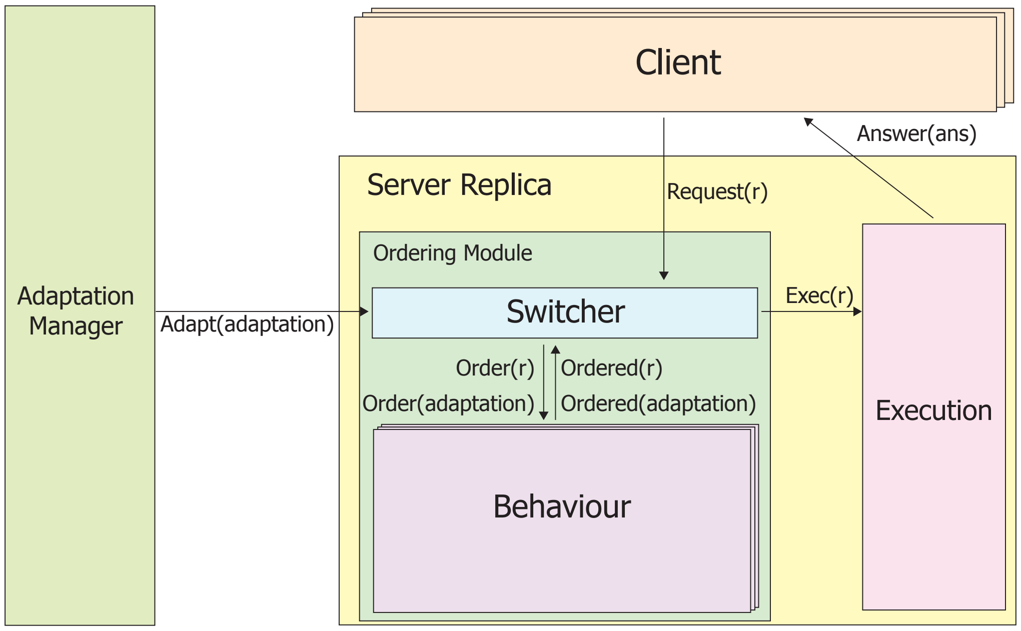


Figure 3.4: Architecture of the developed system to support adaptation in BFT-SMaRt.

Besides the new consensus protocol and the switcher module, other extensions were developed for BFT-SMaRt, namely, an extended dispatch layer, finer grained adaptations and stoppable versions of Mod-SMaRt and Fast-SMaRt. The dispatch layer was modified so it could deal with messages from different protocols, so it can function properly when more than on protocol in execution. Other finer grain adaptations, like leader switching and logging toggle, were developed both to provide support to experiments in the context of the whole Abyss system and to make possible the comparison between reconfiguring a single protocol *versus* switching between protocols with different configurations.

A representation of the architecture is presented in Figure 3.4

3.3.3 Fast-SMaRt

Fast-SMaRt was implemented by extending the classes that modelled the behaviour of replicas and messages for the consensus protocol already existent for Mod-SMaRt. So, new classes that provided the desired behaviour for Proposers, Acceptors and Learners in the Fast Byzantine Consensus protocol. A new class of messages was also developed so it could carry the information needed by the protocol.

Moreover, were performed some modifications to the *TOMLayer* class, which is responsible for sending ordered requests to execution, among other things. This modifications made possible that a request is sent for execution a step earlier than in the original BFT-SMaRt consensus protocol. The dispatch layer that handled incoming consensus messages was also modified so it could support non-authenticated messages because Fast Byzantine Consensus does not make use of this in order to boost its performance.

3.3.4 Safe-SMaRt

Safe-SMaRt was derived from the ideas of Aardvark in the sense of better tolerating faulty client behaviour. Specifically, this variation of Mod-SMaRt detects and deals with clients not operating in a closed-loop (waiting for the answer of request n before sending request $n+1$), as in the BFT-SMaRt protocol this is the correct way of operating. As there are no dedicated channels of communication for each client, like in Aardvark, it is not possible to shut the communication completely. So, a simpler version of this was done by ignoring faulty clients' requests, not putting it in the ordering queue. This way the queue is not filled with faulty requests nor consensus runs are spent ordering them, thus protecting the performance of the system.

To detect such behaviour, every time a client submitted a new request, the event is registered in a table. This register is removed when the said request is ordered and goes to execution. So, if a client submits a request and another request is already registered, it indicates that the client is probably faulty and the request is ignored. Nevertheless, the request is not totally discarded, it is put on a quarantine queue, because it can happen that the replica identifies that the client is on an open-loop when actually it is the replica itself that is behind in the history of executions. Therefore, when a state transfer happens and the replica notices that it was delayed and the quarantined request were correct, it puts them on the queue and removes the register of in-progress quests.

Note that this was developed only as an artefact to perform some experiments and does not aim at making the system as robust as possible against faulty clients. In fact, this is a solution that deals with a very specific vector of attack and still doesn't solve the problem totally, only mitigates it partially. For example, if the attack is performed at the TCP level, this solution does not provide any benefit.

3.3.5 Support for Parallel Execution of Protocols

To allow for a parallel execution of protocols, both in the cases of parallelization optimizations and the parallel execution that happens after an adaptation using non-reconfigurable protocols, the dispatch layer was extended. A new identification field was added to the consensus messages in order to distinguish which protocol they belong to.

In practise this could be done just by identifying the class of the incoming message, as each protocol present in the system has different classes for their messages. Nevertheless, if one opts to use the same protocol but with different configurations, this solution becomes insufficient. Because of this, each consensus message has a identification field that identifies from which configuration it is, expanding the concept of having only different protocols to having distinct configurations, regardless of the actual protocol it belongs to.

Moreover, the log of operations had to be incorporated in the switcher responsibilities. This way, the switcher manages the overall log, of the operations that contributed to the history of the service, by including the operations of active protocols. Nevertheless protocol-specific logs must be kept to allow the tentative execution of protocols that don't write right away for the history of the progress of the system.

Currently, all the configurations (available combinations of protocols with their parameters) must be instantiated at the start of each replica execution. They are declared in the code that runs at the start of the system replica. It would be more flexible if these configurations could be added at run time, but unfortunately this was not implemented due to time constraints.

3.3.6 Adaptation Requests

The adaptation manager's replicas send adaptation commands via socket to the switchers. This command is a string which includes a sequence number of the adaptation to be performed, a description of the adaptation (with type and arguments) and an authenticity proof, which denotes that the adaptation was, in fact, issued by the adaptation manager. This proof is a signature of the previous content, using RSA with SHA256 hashing. This is an expensive computation, nevertheless, as the adaptation requests are not very frequent, it does not represent a threat to the system's performance. The adaptation command is formatted in the following manner:

<sequence number>:<adaptation type>[:adaptation arguments]*:proof

Each switcher, upon receiving a quorum of similar adaptation requests, builds an object containing all the parameters of the configuration, as well as a concatenation of all the proofs (to prove the rightfulness of the adaptation to other replicas). This object is similar to an ordinary client request so it can be ordered in the same way, having only a flag identifying it as a adaptation, so it is recognized by the switcher or the reconfigurable protocol.

3.3.7 Adaptation Execution Flow

An informal description of the flow of introducing an adaptation on the developed system is presented below:

1. Each replica awaits for the collection of n similar and properly signed reconfiguration request from the adaptation issuer replicas. This number is the minimum number of equal messages to tolerate the modelled Byzantine behaviour form the adaptation issuer. For example, if on wants to tolerate up to one failure in the adaptation issuer replicas and it's assumed that no correct replica can issue a wrong reconfiguration, 2 similar reconfiguration commands shall suffice because by definition at least one is correct. So, generally, $f + 1$ similar commands.

2. At each replica, a reconfiguration command is built, containing a sequence number, a description of the adaptation and a cryptographic proof of the authenticity of the adaptation. This message is broadcast to all replicas mimicking a client request.
3. This message is ordered and the ordering of further requests is halted temporarily. While the system is halted, this message is analysed at every replica and if it proves itself to be sound (authentic and has the right sequence number), the described adaptation is executed.
4. After the execution of the adaptation the ordering of new requests continues normally, with the new configuration installed on the system.

3.4 Implementing Adaptation Using BFT-SMaRt as a Black-Box

A purely black-box prototype was also developed. This system aimed at switching between different protocols running in independent processes, where no knowledge about their functioning is needed, besides the interface. This solution isolates the switcher in an independent process. This process acts as a server proxy, with the execution module co-located with it. It acts in the following manner:

1. Receives the client request with the same interface as a regular BFT-SMaRt server.
2. It sends the request to be ordered in the active protocol (process), or to several, if parallel execution is in place.
3. It collects the ordered requests and sends them for execution, which then answers to the client.

The adaptation requests follow a similar execution flow and when it is executed it changes to which processes incoming requests are sent. All the processes are co-located in the same machine to avoid the increased cost of two network communication steps for each request. A representation of the architecture is present in figure 3.5.

This prototype was not developed fully because it failed to deliver acceptable performance in the proof of concept experiments done. Even in normal operation the system revealed itself to be 3 to 4 times worse in terms of throughput when compared to the solution where the switcher was located in the BFT-SMaRt server process. This loss of performance could derive from several things, for example the cost of inter-process communication, the cost of context switch when switching processes or the cost of Java garbage collector as much more objects are instantiated in this solution. No further experiments were carried out due to time constraints.

Summary

This chapter started by providing a conceptual framework devised for facilitating the reasoning about different adaptation techniques. It divides those techniques into three main categories: using reconfig-

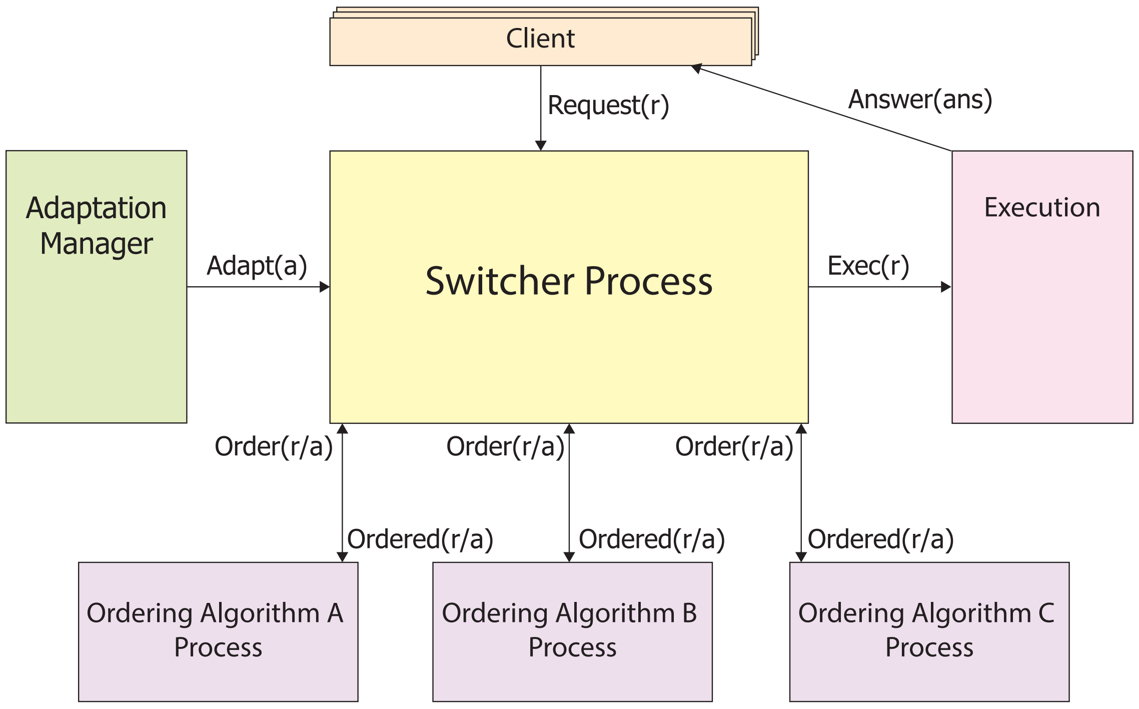


Figure 3.5: Architecture of a switching system using ordering protocols as independent, black-box, processes

urable protocols, usage of protocols modelled as black-boxes and using stoppable protocols. Each category presents a different level w.r.t. the trade-off ease of implementation *vs* potential performance. Reconfigurable protocols promise the best performance but are the hardest to develop, opposing the black-box approach which is the easiest to deploy while the expect performance is lower, finally the stoppable approach sits in the middle of the previous two. Furthermore, some possible optimization are also presented, namely the prioritization of adaptation commands and using parallelization.

We have discussed a number of changes that need to be performed to previous work to support Byzantine Fault Tolerance. Among them, the proofs of adaptation, to mitigate unwanted reconfigurations introduced by Byzantine components. It was also discussed the need for proofs to start a parallel execution of a protocol, to avoid its unrightful usage.

Finally, the implementation of such techniques in the BFT-SMaRt framework has been presented. Some new modules, like the switcher, a new consensus protocol, and a safer mode of operation were described. Furthermore, some extensions were also discussed, namely the support for multiple protocols running concurrently, were also discussed.

4 Evaluation

In order to compare the different adaptation techniques, in this section we intend to answer the following questions:

a) How much time does it take for a reconfiguration to be executed with each technique?

b) What is the impact on the system's performance caused by adapting with the different techniques?

c) What load is imposed by the adaptation on the network?

d) How useful are the the distinct adaptation techniques to bring the system out of a poor performance situation caused by environmental conditions (reactive adaptation)?

e) How large is the advantage of using reconfigurable protocols, instead of stoppable or black-box, if one wants to perform adaptations that do not demand coordination (known as flash adaptations)?

So, in this chapter an experimental evaluation is presented to answer such questions. In Sections 4.1 and 4.2 we address questions a), b) and c). Question d) is discussed in Section 4.4 and question e) is explored in Section 4.3

To mitigate artefacts in the data caused by unwanted hidden variables, like the physical network conditions or the virtual machines performance, 15 runs were performed for each experiment. The 4 most deviating datasets, the two with smaller values and the two with bigger values, were discarded. After that an average was calculated, being the values presented in the graphs. The standard deviation fell between 7% and 12%, nevertheless the differences among the different techniques kept consistent within each of the individual experiment run.

4.1 Local Network Context

To execute this experiments 6 and 11 BFT-SMaRt replicas were used as this is the minimum replicas needed to tolerate one and two faults, respectively, using Fast-SMaRt [1], and a client capable of introducing variable load in the system. All the replicas and the client were hosted in independent virtual

[1]This number of replicas does, in fact, penalize the performance of Mod-SMaRt, has it uses more replicas than necessary. However, we chose not to introduce and remove replicas during our experiments so that the only change happening is the protocol switch. Moreover, we do not intend to prove the efficiency either of Mod-SMaRt nor Fast-SMaRt, but the efficiency of the switch algorithms.

machines on the DigitalOcean[2] cloud provider. Each machine has a dual core CPU running at 2.40GHz, 2GB of RAM and a *full-duplex* 1Gb/s network connection. This configuration was chosen for being the most powerful, hence closer to a real-world server, within the available resources. The client which generates load sends requests in multiple threads, simulating multiple clients. In this section of experiments the client sends a request of 10kB each after receiving the reply (with 10B) of the previous request (works in a *closed-loop*). The system ran for 4 minutes, which were dropped, before any adaptation was executed so that the load introduced by the client could put the system in a stable point of performance.

4.1.1 Adaptation Time

To evaluate the time needed to perform an adaptation using each technique, it was accounted the delay between the arrival of an adaptation request[3], and the application of such adaptation. This has been performed with different amounts of load introduced in the system. The results are presented in Figure 4.1. In the graph it is also represented, as a baseline, the time of a consensus run to order a request, since it represents the minimum time to execute an operation on the system with the given conditions. It is observable that the time of adaptation using reconfigurable protocols grows much slower than the other approaches, following closely the consensus time. This happens mainly because in a solution using reconfigurable protocols it's possible to have access to the queue of incoming requests and prioritize adaptation requests, while in all other techniques that is not possible. This way, when using reconfigurable protocols, the adaptation time grows only with the time taken to order a request, while on the other hand, in the other techniques it grows both with the time to process a request and the queuing time.

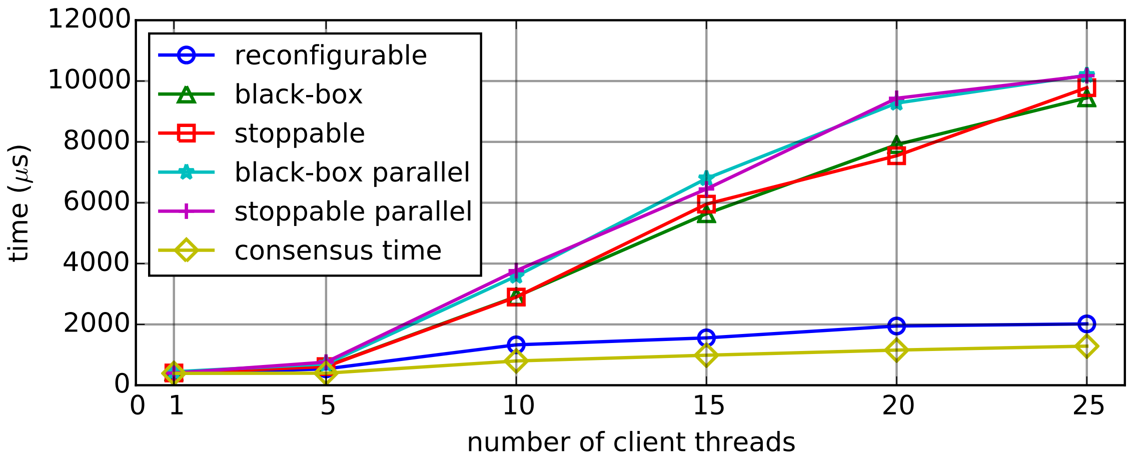


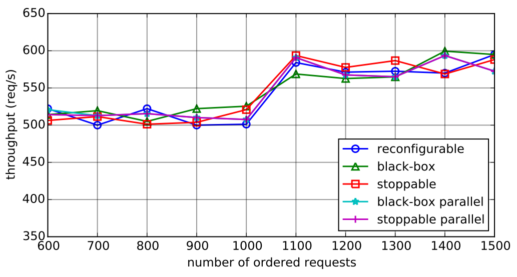
Figure 4.1: Execution time of an adaptation in a local network, using 6 replicas

[2]https://digitalocean.com

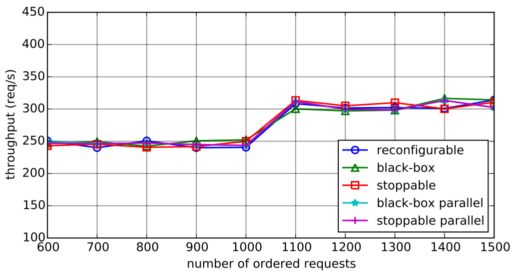
[3]More precisely, the arrival of the second similar adaptation request, in order to have a quorum of commands tolerating one fault in the adaptation manager's replicas.

4.1.2 Throughput During an Adaptation

To measure the overhead of executing an adaptation with each of the techniques described, it was used a client with 20 threads, being this the observed configuration that introduced a significant load in the system without putting it under excessive stress. The throughput was extrapolated at every 100 requests ordered, by measuring the time it took to order the said requests. After the request number 1000 an adaptation request, to switch from Mod-SMaRt to Fast-SMaRt, was submitted to the system. The obtained results are shown in Figure 4.2.



(a) With 6 replicas



(b) With 11 replicas

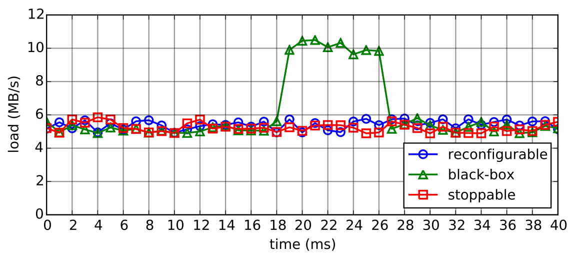
Figure 4.2: Throughput during an adaptation in a local network

It can be observed that, in the context of this experiments, none of the used techniques introduced a visible overhead in the throughput of the system. The rise in throughput between requests 1000 and 1100 exists because the adaptation applied to the system contributed to an overall increase in performance.

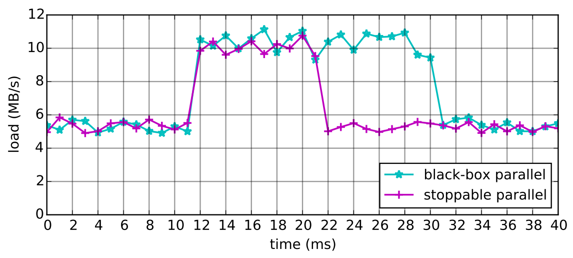
4.1.3 Network Overhead of an Adaptation

The network load was measured during the same experiment described above, with samples every millisecond. In order to facilitate the comparison of the data, the most relevant 40ms of execution were taken and aligned so the reconfiguration request arrives at 10ms. The data is represented in Figures 4.3 and 4.4.

As opposed to throughput, the load introduced in the network varies among the different techniques used. The use of non reconfigurable protocols imposes an increase on network usage after an adaptation is executed as the protocol that ceased to be active continues to execute in the background until it depletes the queue of received requests. When using parallelization techniques, there is a visible overhead on the network prior to the execution of a reconfiguration. This happens because as soon as an adaptation request is received the next protocol to be active starts executing tentatively in parallel with the currently active protocol, existing a period with an increase of nearly 90% of the load on the network, when comparing with the normal execution.



(a) Techniques without parallelization



(b) Techniques with parallelization

Figure 4.3: Network load during an adaptation, in a local network, using 6 replicas

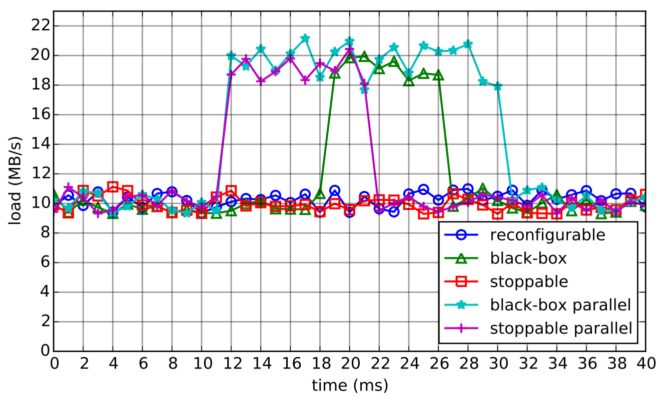


Figure 4.4: Network load during an adaptation, in a local network, using 11 replicas

4.2 Wide-area Network Context

To experiment the different techniques in an wide-area network context, a cross-datacenter network was emulated. To emulate the said network topology, 6 replicas were hosted in the same datacenter and latency was introduced between them at the Linux Kernel level, using the *netem*[4] tool. To simulate a network environment as close as possible to a realistic use case, real latency values across Amazon datacenters were used. The emulated network simulates having a system replica in each of the following datacenters: North California (0), Oregon (1), Ireland (2), Frankfurt (3), Tokyo (4) and Sidney (5). The amount of latency between each replica is presented in (Bravo, Rodrigues, and Van Roy 2017) and is shown in Table 4.1. To emulate the variability in the communications delay, a jitter of $\pm 10\%$ of the latency added.

4.2.1 Adaptation Time

Except from the network environment, this experiment closely followed the one described in section 4.1.1. However, due to the increase of latency, the batching mechanism of BFT-SMaRt was more prominent, reducing the load of consensus messages to be processed. This raised the need to introduce more clients to induce enough load to bring the system performance to its peak. So, 2 multi-threaded

[4]https://wiki.linuxfoundation.org/networking/netem

Table 4.1: Latencies between the different replicas in the system used in the wide-area network experiments

Replica	1	2	3	4	5
0	10ms	74ms	84ms	52ms	89ms
1	-	69ms	79ms	45ms	81ms
2	-	-	10ms	107ms	154ms
3	-	-	-	118ms	161ms
4	-	-	-	-	52ms

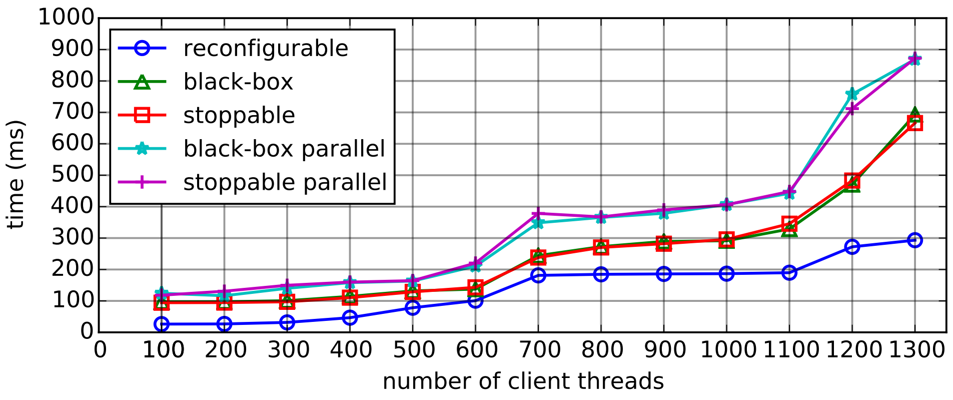


Figure 4.5: Reconfiguration time in a wide-area network environment

clients were used with a total number of threads between 100 and 1300, with half of the threads running in each client process. The collected data is shown in Figure 4.5.

The results show that the difference in time between adapting using a reconfigurable protocol *versus* adapting with other techniques is much closer to be constant than in a local network, mainly up to the load introduced by 1100 replicas. This happens because the latency introduced by the network (felt by all the techniques) is much more relevant than the latency introduced by the queuing time, which is the main differentiator of the distinct techniques reconfiguration times. Nevertheless, the absolute value of the difference between the distinct solutions is much more prominent than in a local network, instead of less than ten milliseconds, in this environment the differences were of hundreds of milliseconds.

The steeper slope observed starting in the load introduced by 1200 clients derives from the fact that the system starts to fail in coping with the number of incoming requests, having greater queuing times, which affects directly the time of executing an adaptation when using non reconfigurable or stoppable protocols.

We can then conclude that in a wide-area network environment, there is a greater load span in which the different techniques grow more similarly, when compared to a local network. However differences exist in the overall time, and may get several orders of magnitude different if the queuing time surpasses the ordering time of a request. This can be of interest if the adaptation is time-sensitive and must be performed as fast as possible, as in a local network, using a reconfigurable protocol is the faster technique.

4.2.2 Throughput During an Adaptation

To test the overhead of carrying an adaptation in an wide-area network environment, a similar method to the one described in 4.1.2 was followed. 6 BFT-SMaRt replicas and 2 multi-threaded clients were used. Because of the higher latency in request processing, it was possible to collect data with time based samples. This is, instead of collecting data samples every 100 requests ordered, like in the local network experimental method, the throughput was calculated at every 100ms based on the number of requests answered in that time. This approach facilitates the interpretation of the results as it's more intuitive to rationalize using the passage of time instead of the number of requests answered. After 4 minutes, 20 seconds of execution were registered, where an adaptation request arrived at the system shortly after the 500th millisecond. The throughput of answered requests during this time, with a load of 1000 and 1200 client threads, is represented in Figures 4.6 and 4.7. This loads were chosen because 1000 clients introduced a steady peak performance and 1200 clients introduced a near-peak performance with an increase in queuing time.

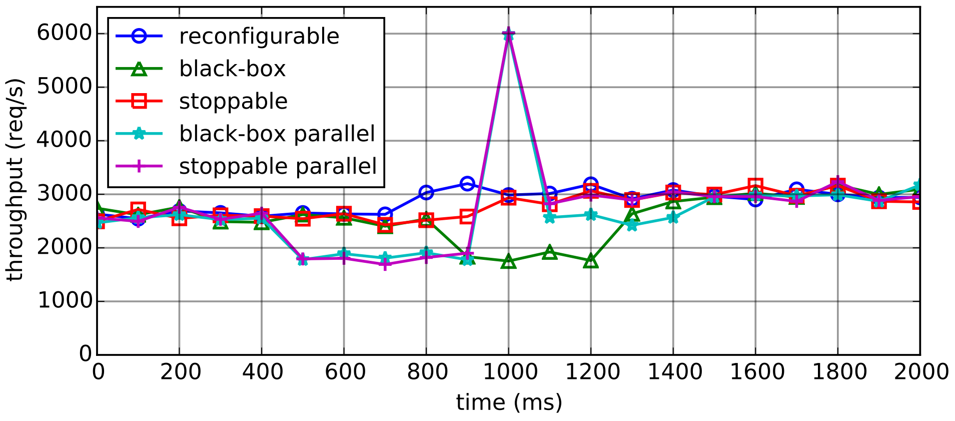


Figure 4.6: Throughput during an adaptation, in an wide-area network, using 1000 client threads.

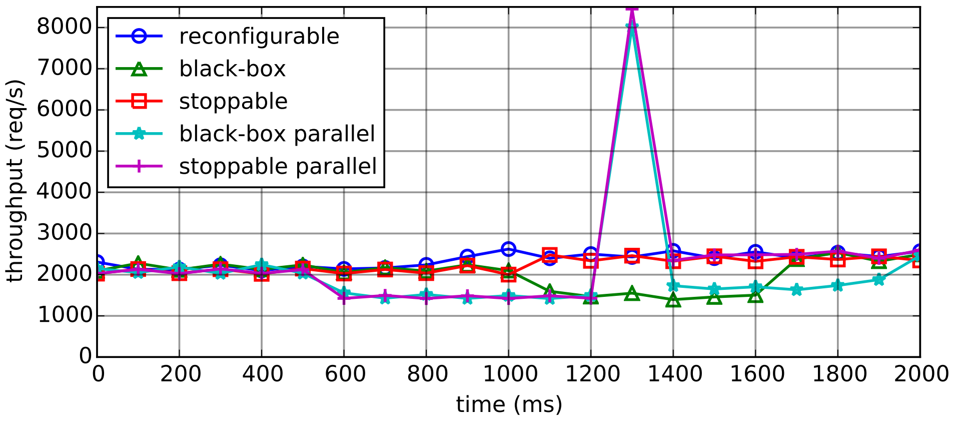


Figure 4.7: Throughput during an adaptation, in an wide-area network, using 1200 client threads.

As the adaptation takes more time in this network environment the differences in performance among the different techniques become more apparent. One can note that the adaptation using reconfigurable and stoppable protocols carry little to no overhead in throughput. On the other hand, solutions using non-reconfigurable protocols carry a penalization in throughput in the moments after an adaptation, because the protocol that ceased to be active continues to operate after the adaptation, until it depletes the existing queue of incoming requests, hence wasting computational resources. Finally the approaches that use parallelization present a belly before the executing of the adaptation and a peak in throughput right after the adaptation is executed. The loss of throughput happens because of the parallel execution of protocols in these techniques, carrying costs in performance due to the sharing of resources among both protocols. The peak happens because the new protocol already started ordering requests as soon as the adaptation command arrived and now it can dispatch in burst all the processed requests to the clients.

The behaviour of parallelization techniques arises to question if the peak reached after the adaptation compensates the loss of performance before it, when compared to their counterpart techniques that use no parallelization. In order to answer this question, a graph showing the total amount of requests answered with the different techniques was derived from the throughput data. The results are presented in Figures 4.8 and 4.9.

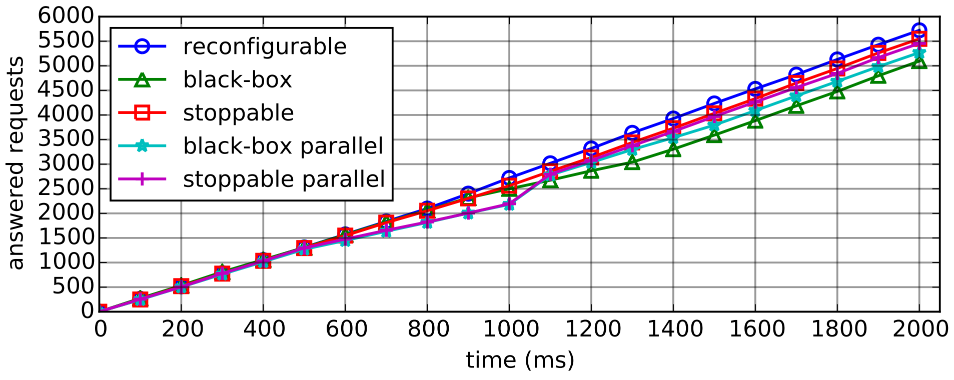


Figure 4.8: Answered requests during an adaptation, in an wide-area network, using 1000 client threads.

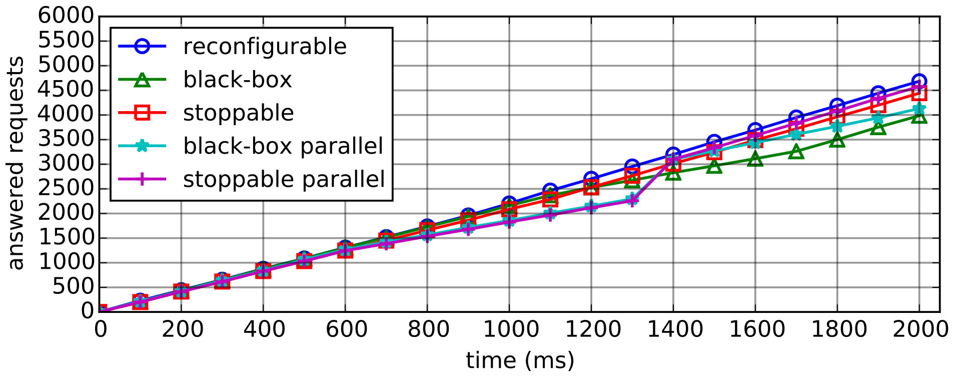
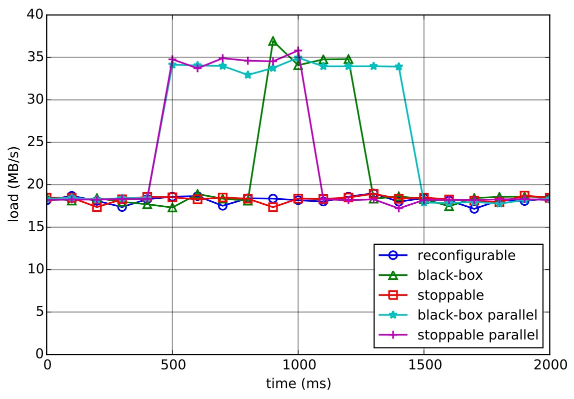
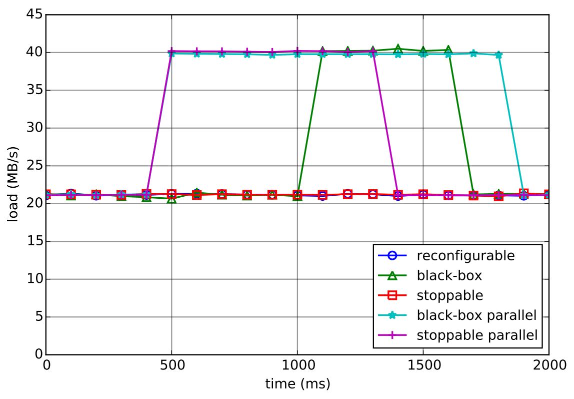


Figure 4.9: Answered requests during an adaptation, in an wide-area network, using 1200 client threads.

The data shows us that in the case of using black-box protocols the parallelization optimization reveals itself to be of worth, compensating, in part, the penalization introduced by the use of black-box protocols when compared to reconfigurable and stoppable ones. Although the difference is only about 100 requests in this experiment, this small differences may add up and impact a long-lived system with



(a) With a load of 1000 clients.



(b) With a load of 1200 clients.

Figure 4.10: Load introduced in an wide-area network when executing an adaptation

recurring adaptations. On the contrary, when comparing the use of parallelization when using stoppable protocols the data shows that not always the parallel execution is beneficial. This happens because using stoppable protocols does not incur in visible losses of performance by itself. So, the issue relies on if the peak of throughput using stoppable protocols with parallelization compensates the loss of performance prior to the adaptation because there is no loss of performance to cover, like when using black-box protocols.

We can see that parallelization surpasses its counterpart when using 1200 client threads, while it doesn't when using 1000 client threads. The main difference among them is the time between receiving an adaptation request and executing it, which is the time that the soon-to-be active protocol executes tentatively. It's visible that with a greater time, the parallelization technique has better results. This happens because the consensus protocols usually have a warm-up time until they reach peak performance, in the Fast-Smart case its due to the batching behaviour, which benefits with queues which length is near the maximum batch size. So, with shorter adaptation times, the time spent warming up the new protocol may not compensate the overall loss of performance due to parallelization. In the conditions of this experiment, a possible optimization for adapting using stoppable protocols and parallelization would be to delay the execution of the adaptation so the protocol executes enough time to compensate the overall drop in throughput, contrary to the intuitive idea of executing an adaptation as fast as possible. Of course this is a very specific case, where the adaptation is being made only to increase performance, not being critical or time sensitive as it would be if it was done for security purposes, for example. Furthermore, the peak throughput of the protocol executing tentatively must be higher than the loss of throughput in the active protocol when compared to the throughput it would have if no other protocol was executing.

4.2.3 Network Load Overhead of an Adaptation

During the experiments described above, the network load was also collected. The results are shown in Figure 4.10. Although the differences are amplified due to greater load and adaptation time, the conclusions are similar to the ones discussed in 4.1.2.

4.3 Adaptation Under Heavy Load

All the previous discussed experimental cases assumed a stable execution of the system, this is, the system was not operating in conditions that aggressively reduced its performance. This leads to the question: *How useful are the the distinct adaptation techniques to bring the system out of a poor performance situation caused by environmental conditions?*

To answer this question, a simulation of several clients becoming faulty, and introducing too much load, was done to test how the adaptations could help dealing with it. Specifically, the experiment simulates clients who operate out of the BFT-SMaRt protocol by executing in an open-loop, this is, not waiting for the answer to request n before sending request $n+1$. To emulate a possible adaptation issued by the adaptation manager, a protocol switch to Safe-SMaRt was issued when the system's latency degraded beyond 43 seconds. This happened at second 0 in the graph presented in Figure 4.11.

The network conditions are similar to the ones used in 4.2. There were used 1000 clients, where half of them, 500, were faulty and operating in an open-loop, ignoring the answers of the replicated server.

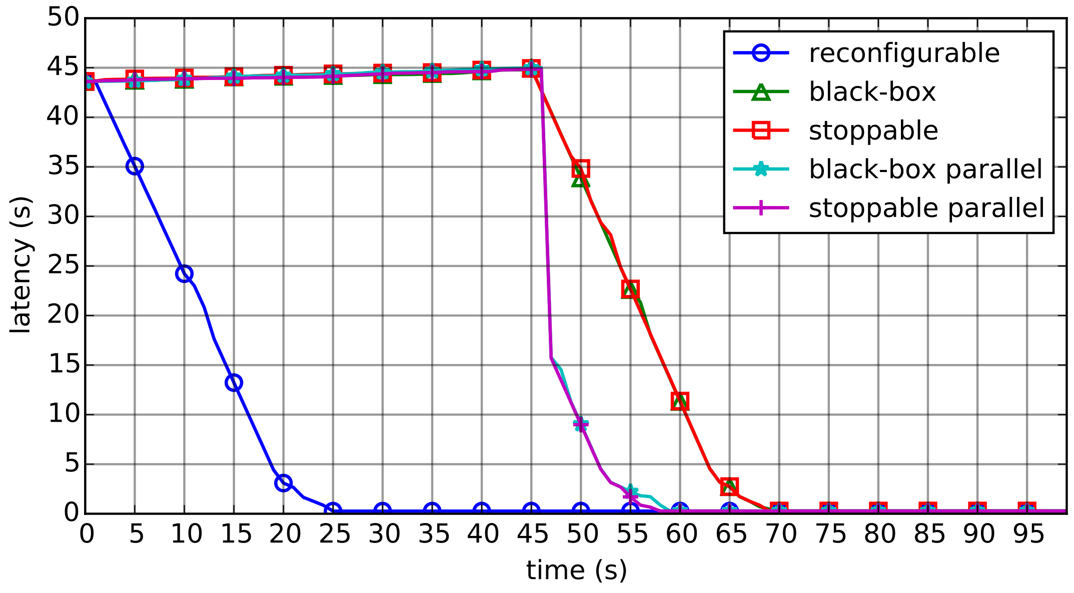


Figure 4.11: Introduction of an adaptation when the system's performance is heavily degraded

Note that this experiment does not aim at proving the utility of Safe-SMaRt improve the resilience of the system to faulty clients, as that falls out the scope of this thesis. This experiment was performed to compare how each adaptation technique performs when used in heavily degraded system's performance conditions.

It is possible to note that using adaptable protocols is again the fastest way to introduce an adaptation, being introduced more than 40 seconds before any other technique in this case. This happens because of the prioritization of adaptation requests, as discussed before. A new insight, not verified clearly before, is that in this case, the parallelization performs better that non-parallel commutation, both for stoppable and black-box adaptation. Even if they are applied some seconds later, their parallel execution helps to bring the system to a good performance much faster. This happens because the new protocol, executing in parallel, has the chance to be operating during a considerable amount of time, being able to operate in peak performance for a great amount of time. Moreover, the new protocol is more resilient to the attack being performed, having better performance than the previous protocol. Note that, although this is verifiable in this case, in some performance degradation contexts, parallelization may be worse than non-parallel techniques, specially if the bottleneck in performance is bandwidth or computing power.

4.4 Flash Adaptation *versus* Ordered Adaptation

As seen in 3.2.1, when using reconfigurable protocols it is possible to introduce fine-grained adaptations. Even more so, some of this adaptations can be performed without the necessity of running a consensus. This arises the question *How great is the advantage of using reconfigurable protocols, instead of stoppable or black-box, if on wants to perform adaptations that don't demand coordination?* To answer this, the delay of introducing an adaptation which does not require a consensus (called flash adaptation) using a reconfigurable protocol *versus* using other techniques, namely stoppable and pure black-box protocols. The adaptation introduced aimed at start logging the replicas activity to disk. A fine-grained adaptation which demands coordination, specifically changing the leader, is also present in the chart to allow a comparison between a coordination-demanding and a flash adaptation using a reconfigurable protocol.

The experimental setup follows closely the one used for wide-area performance experiments (Section 4.2), but with added latency between the adaptation manager's replicas and the system's replicas. This latency was added using netem, as before, with values between 25ms and 35ms. The time was measured between issuing the first adaptation command at the adaptation manager's replicas and it being applied by a quorum of system's replicas, which consists in 5 replicas in this context. The last replica was ignored as it could be faulty, under the system model. The obtained results are presented in Figure 4.12.

It is visible that flash adaptations, using reconfigurable protocols, are applied much faster than when the same adaptation is applied with other techniques. This follows what was expected, as when using stoppable and black-box protocols this adaptations still demand running consensus to switch between two protocols with different configurations, whereas when using reconfigurable protocols it does not.

The majority of the time taken by the flash technique to apply the configuration is actually the latency between the adaptation manager's replicas and the switchers, as applying the adaptation is in practice achieved with just some simple method calls. On the other hand, when applying a coordination-demanding adaptation, it takes more time due to the need of a consensus run. The differences among the other techniques derive from the loss of performance when using parallelization and the queueing

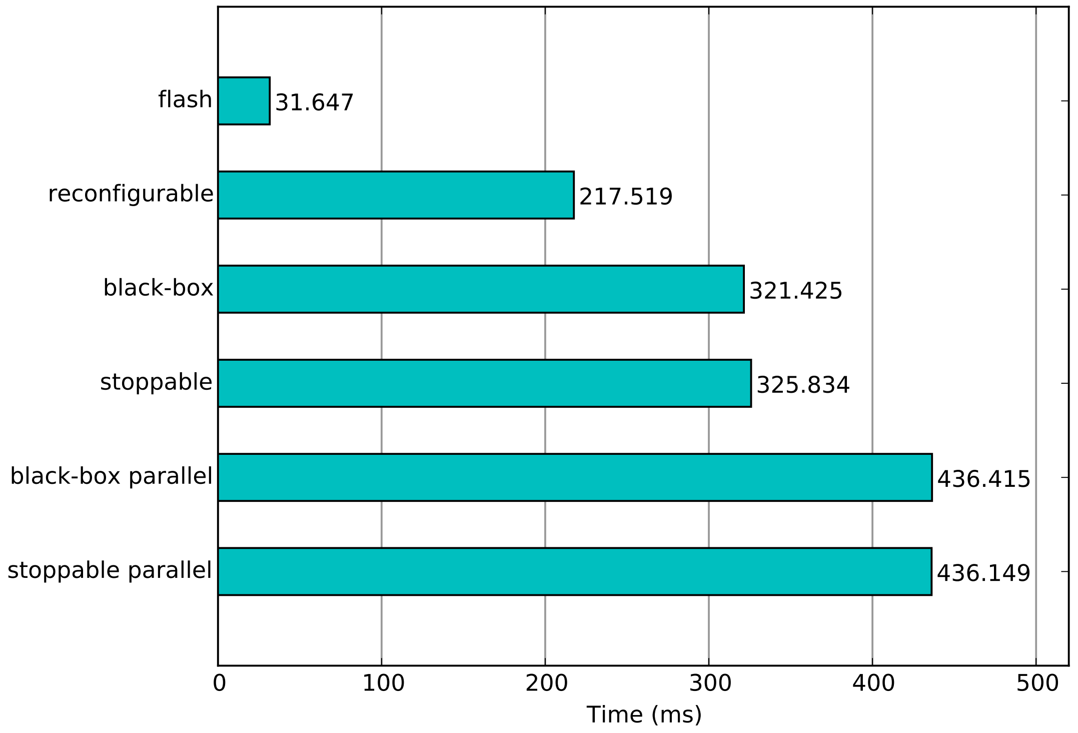


Figure 4.12: Adaptation time of flash, coordination-demanding, stoppable and pure black-box approaches.

time, as already previously discussed in 4.2.

Summary

This chapter presented experimental insights of how the different adaptation techniques performed in practise. As it was shown, in a local network the overhead caused by reconfiguration was negligible, no matter what technique was used. Therefore, switching between protocols that behave as black-boxes may be the best solution, given that this is the easiest reconfiguration technique to deploy. On the contrary, in wide-area networks, switching between black boxes penalises the throughput. Hence, using stoppable protocols appears to be the best solution for this scenario.

On the other hand, when reducing the delay of the adaptation is critical, the usage of reconfigurable protocol is advised, as this allows for the fastest transition. This is mainly due to the possibility of performing a prioritization of the adaptation commands. This is specially relevant if one wants to use adaptation to bring the system out of a heavily degraded configuration. Reconfigurable protocols also allow for the fast execution of fine-grained adaptations, while the other techniques do not.

Parallelization revealed itself to be useful if one uses the black-box approach in an wide-area context. Its usage with stoppable protocols is only beneficial if the adaptation takes a significant amount of time to be applied, due to warm-up time of the protocols. It may be specially relevant if it is applied in a heavily degraded performance state.

5 Conclusions

In this dissertation we have addressed the problem of performing dynamic reconfiguration of BFT systems. We have organized, in a systematic manner, a portfolio of algorithms to switch in run-time between different protocols. The algorithms have been derived by adapting previous solutions that have been developed for different fault models.

The algorithms can be applied in different scenarios, depending of the properties of the protocols involved in the reconfiguration. We have classified the target protocols into three main categories, namely: reconfigurable protocols, stoppable protocols, and protocols without any specific support for adaptation, that need to be treated as black-boxes. We have also identified several optimizations that can be applied to the reconfiguration algorithms, such as prioritizing adaptations and using parallelization.

To understand the tradeoffs involved when applying these algorithms in practice, and to obtain a comparative assessment of their performance, we have implemented them in a common framework, based on the BFT-SMaRt open source project. We have deployed these implementation in different settings.

Our results show that in a local network, using target protocols as a black-box reveals itself to be the best solution given that it presents the same performance as other alternatives and it is the easiest to deploy. On the contrary, when performing reconfiguration in a geo-replicated system, the use of black-box protocols presents a significant overhead when compared to other alternatives. Thus, it may be worth to change to target protocols such that they support at least a stoppable interface. We have also observed that parallelization is always useful for high-bandwidth networks when using black-box algorithms. However, if stoppable algorithms are used, parallelization is only beneficial in face of large network delays. Finally, as expected, if the target protocols are reconfigurable, switching can be executed with significant savings, namely in terms of latency.

As future work we would like to use the findings reported here to improve adaptation policies for several concrete applications that can benefit from dynamic reconfiguration, such as geo-replicated distributed ledgers. Another interesting direction in future work is to combine different switching algorithms in order to switch between protocols with distinct levels of support for adaptation.

References

Aublin, P., R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić (2015, January). The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS) 32*(4), 12:1–12:45.

Bessani, A., J. Sousa, and E. Alchieri (2014, June). State machine replication for the masses with bft-smart. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, USA, pp. 355–362. IEEE.

Bortnikov, V., G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman (2015, December). Reconfigurable State Machine Replication from Non-Reconfigurable Building Blocks. *ArXiv e-prints*.

Bracha, G. and S. Toueg (1983, August). Resilient consensus protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Montreal, Quebec, Canada, pp. 12–26. ACM.

Bravo, M., L. Rodrigues, and P. Van Roy (2017). Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, pp. 111–126. ACM.

Carvalho, C., D. Porto, L. Rodrigues, and A. Bessani (2017, October). Adaptação dinâmica de protocolos de consenso bizantino. In *Actas do nono Simpósio de Informática (Inforum)*, Aveiro, Portugal.

Castro, M., B. Liskov, et al. (1999, February). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 173–186.

Chen, W., M. A. Hiltunen, and R. Schlichting (2001, April). Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pp. 635–643. IEEE.

Clement, A., M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche (2009, October). Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pp. 277–290. ACM.

Clement, A., E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti (2009, April). Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, pp. 153–168. USENIX Association.

Copeland, C. and H. Zhong (2014). Tangaroa: a byzantine fault tolerant raft. Technical report, Secure Computer Systems Group, Stanford University.

Couceiro, M., P. Ruivo, P. Romano, and L. Rodrigues (2015, November). Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *IEEE Transactions in Parallel and Distributed Systems 26*(11), 2942–2955.

Diffie, W. and M. Hellman (1976, November). New directions in cryptography. *IEEE transactions on Information Theory 22*(6), 644–654.

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

Guerraoui, R. and L. Rodrigues (2006). *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Kotla, R., L. Alvisi, M. Dahlin, A. Clement, and E. Wong (2007, October). Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 45–58. ACM.

Lamport, L. et al. (2001, December). Paxos made simple. *ACM Sigact News 32*(4), 18–25.

Lamport, L., D. Malkhi, and L. Zhou (2008). Stoppable paxos. Technical report, Microsoft Research.

Lamport, L., D. Malkhi, and L. Zhou (2010, March). Reconfiguring a state machine. *ACM SIGACT News 41*(1), 63–73.

Lamport, L., R. Shostak, and M. Pease (1982, July). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*(3), 382–401.

Lorünser, T., A. Happe, and D. Slamanig (2014). Archistar - a framework for secure distributed storage. `http://ARCHISTAR.at`. GNU General Public License.

Martin, J. and L. Alvisi (2006, July). Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput. 3*(3), 202–215.

Mocito, J. and L. Rodrigues (2006, August). Run-time switching between total order algorithms. In *Proceedings of the Euro-Par 2006*, LNCS, Dresden, Germany, pp. 582–591. Springer-Verlag.

Ongaro, D. and J. Ousterhout (2014, June). In search of an understandable consensus algorithm. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, Philadelphia, PA, USA, pp. 305–319. USENIX Association.

Rosa, L., L. Rodrigues, and A. Lopes (2007, October). A framework to support multiple reconfiguration strategies. In *Proceedings of the Autonomics (AUTONOMICS)*, Rome, Italy, pp. 15.

Sabino, F. (2016, September). Bytam: a byzantine fault tolerant adaptation manager. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Sabino, F., D. Porto, and L. Rodrigues (2016, September). Bytam: um gestor de adaptação tolerante a falhas bizantinas. In *Actas do oitavo Simpósio de Informática (Inforum)*, Lisboa, Portugal.

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

Singh, A., T. Das, P. Maniatis, P. Druschel, and T. Roscoe (2008). Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Volume 8, pp. 189–204. USENIX Association.

Sousa, J. and A. Bessani (2012, May). From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Proceedings of Ninth European Dependable Computing Conference (EDCC)*, pp. 37–48. IEEE.