ByTAM: a Byzantine Fault Tolerant Adaptation Manager

Frederico Miguel Reis Sabino

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Doutor Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson:	Prof. Doutor João António Madeiras Pereira
Supervisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member of the Committee:	Prof. Doutor Alysson Neves Bessani

September 2016

Acknowledgements

I wish to thank Daniel Porto for his insightful suggestions and invaluable help for understanding the internals of BFT-SMaRt. I am also grateful to Prof. Rodrigo Rodrigues and Prof. Miguel Correia for their comments on the early stages of this work. Finally, I wish to thank the BFT-SMaRt team for their support.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and by PIDDAC through projects with references PTDC/EEI-SCR/1741/2014 (Abyss) and UID/CEC/50021/2013.

Lisboa, September 2016
Frederico Miguel Reis Sabino

For my family and friends,

Resumo

Embora existam já algumas propostas para adaptação dinâmica de protocolos tolerantes a faltas bizantinas (BFT), as soluções atuais tem limitações importantes como pouca robustez, dependendo da existência de componentes centralizados e confiáveis; pouca flexibilidade, restringindo a estratégia de adaptação; e falta de extensibilidade, impossibilitando a adição de novas políticas. Para superar estas lacunas, esta dissertação propõe um gestor de adaptação genérico, que pode ser usado para controlar a execução de diversas políticas de adaptação, permitindo ajuste fino dos parâmetros dos protocolos. Este gestor funciona de forma independente do serviço que se pretende adaptar, sendo ele próprio tolerante a falhas bizantinas, e foi desenvolvido recorrendo a uma conhecida biblioteca de código aberto (BFT-SMaRt[1]). A sua avaliação experimental mostra que este é eficaz na adaptação, promovendo um desempenho mais eficiente do serviço BFT em diversas condições de operação.

[1] https://github.com/bft-smart/library

Abstract

Previous systems that support the dynamic adaptation of Byzantine Fault Tolerant (BFT) protocols have important limitations such as: lack of robustness (dependence on central and trustworthy components); low flexibility (constraining the adaptation strategies); and lack of extensibility (making it impossible to add new policies). To overcome these gaps, this dissertation proposes a generic BFT adaptation manager which can be used to execute different adaptation policies. This manager is independent of the target service which needs to be adapted dynamically and is, by itself, also tolerant to Byzantine Faults. The manager has been implemented using BFT-SMaRt[2], a well known open-source library. The experimental evaluation of the resulting prototype illustrates that it can be used to effectively increase the performance of a target BFT service in different operational conditions.

[2]https://github.com/bft-smart/library

Palavras Chave
Keywords

Palavras Chave

Tolerância a Falhas Bizantinas

Adaptação Dinâmica

Políticas de Adaptação

Reconfiguração Activa

Replicação de Máquinas de Estado

Keywords

Byzantine Fault Tolerance

Dynamic Adaptation

Adaptation Policies

Active Reconfiguration

State Machine Replication

Contents

List of Figures

List of Tables

Acronyms

BFT Byzantine Fault Tolerance

JVM Java Virtual Machine

KVM Kernel-based Virtual Machine

PBFT Practical Byzantine Fault Tolerance

API Application Program Interface

1 Introduction

State machine replication with Byzantine Fault Tolerant (BFT) protocols is a technique that allows building robust services, capable of working correctly, even in the presence of a minority of arbitrary faults (accidental or malicious), caused by external attacks or intrusions. Essentially, these protocols implement a solution of *consensus*, a fundamental problem in distributed systems that has been extensively studied in literature(Cachin, Guerraoui, and Rodrigues 2011; Fischer, Lynch, and Paterson 1985; Lamport, Shostak, and Pease 1982).

In this dissertation, we present ByTAM, a generic adaptation manager which allows the dynamic adaptation of a given target BFT protocol. ByTAM supports custom policies which can be dynamically selected, allowing it to execute any adaptation policy which can be expressed using rules of the type "condition-event-action". ByTAM is tolerant to Byzantine faults, where the adaptation manager and the associated monitor infrastructure operate independently of the service being replicated and adapted. The resulting architecture, which allows the fine tuning of the parameters of the target BFT protocol, requires few modifications to the protocols being adapted. This feature eases the incremental transition of legacy systems (non adaptable) to new systems with dynamic adaptation capabilities. Furthermore, ByTAM is capable of dealing with issues such as failures and asynchrony which, when unhanded, could prevent the different replicas of the adaptation manager to make consistent decisions. ByTAM is an open source project which is integrated with BFT-SMaRt (Bessani, Sousa, and Alchieri 2014), a library used for the development of BFT systems.

1.1 Motivation

BFT is required to operate correctly in face of faults but it must also exhibit good performance. In fact, the performance of the protocol also has a strong impact on the final user experience. Most BFT protocols are optimized to achieve the best performance on the most common operational conditions (eg. local networks or WAN, sporadic failure events, etc.). As a result, these protocols may be penalized when operating outside the expected conditions. For instance, Zyzzyva(Kotla, Alvisi, Dahlin, Clement, and Wong 2007) when compared with PBFT (Castro, Liskov, et al. 1999), performs better when the threat level (failures or presence of Byzantine processes) is low. However, Zyzzyva's complex strategy to recover from failures ends up being less efficient than PBFT in scenarios where failures are more frequent(Singh, Das, Maniatis, Druschel, and Roscoe 2008). Furthermore, the factors that affect the performance of the protocol such as the number of replicas, the size of the clients' requests or the threat level, have values that may change overtime.

Thus, it becomes relevant to use techniques that allow the dynamic adaptation of the parameters

and the algorithms used by the BFT services. Some protocols, such as Aliph(Guerraoui, Knežević, Quéma, and Vukolić 2010) and ADAPT(Bahsoun, Guerraoui, and Shoker 2015) already provide some support towards this goal, allowing a protocol change if a failure occurs or if a certain condition is not met, combining strategies of different BFT algorithms. Still, these solutions are not flexible regarding the adaptation mechanisms they support, the policies which trigger the adaptation, the adaptation strategy (change of protocol, execution cancellation), and the degree of robustness (centralized components or non-BFT). Despite the merit of these pioneer works, it is still necessary to make significant progress to derive practical solutions, which are simultaneously robust and efficient. The generic adaptation manager proposed in this dissertation addresses these limitations.

1.2 Contributions

In this dissertation we describe the design, implementation and evaluation of a generic adaptation manager which allows the dynamic adaptation of a given target BFT-system. More specifically this dissertation makes the following contribution:

It proposes an architecture that supports the fault-tolerant implementation of a closed control loop that: i) monitors the environment, to collect information regarding the factors that affect the performance of the target BFT-system; ii) executes user defined policies to derive the required adaptations and; iii) executes the selected adaptations. All activities of the control loop are implemented by modules designed to tolerate Byzantine-faults.

1.3 Results

The results of the dissertation are:

1. ByTAM, an implementation of the proposed architecture, including an Byzantine-tolerant adaptation manager implemented as an extension of BFT-SMaRt(Bessani, Sousa, and Alchieri 2014).
2. An experimental evaluation of ByTAM.

1.4 Research History

We have decided to use BFT-SMaRt as the basis to the implementation of ByTAM because it is one of the few open-source BFT implementations which is still actively maintained by its creators. Furthermore, the fact the the BFT-SMaRt team is located in Lisbon, helped to resort to their help when faced with problems with the current BFT-SMaRt code. During my work, I benefited from the fruitful collaboration of Daniel Porto, who provided many invaluable contributions to the design and implementation of ByTAM. Previous descriptions of this work were published in (Sabino, Porto, and Rodrigues 2016).

1.5 Structure of the Document

The rest of this document is organized as follows. Section 2 addresses the related work and background of some BFT systems. Chapter 3 presents the design of ByTAM, highlighting its architecture components and their main functions. Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

In this section we make an overview of the work on Byzantine Fault Tolerance. In Section 2.1, we start by discussing how Byzantine faults relate to other types of faults typically considered in fault-tolerant systems. Then, in Section 2.2, we describe the model that characterizes the system we are targeting. We then proceed to enumerate the properties that a BFT protocol must preserve in Section 2.3. In Section 2.4 we describe some of the most relevant BFT protocols that have been proposed in the literature.

2.1 Fault Model

We consider a distributed system composed of n nodes, out of which f may be faulty. The relationship between n and f is usually called *resilience* (Cachin, Guerraoui, and Rodrigues 2011) and depends on the type of faults that one aims at tolerating (and also on the system model, described in the next section). Not surprisingly, some faults are easier to tolerate than others. A possible classification of faults, in increasing order of severity, is captured by the following list of fault types (Cachin, Guerraoui, and Rodrigues 2011):

- *Crash-stop*: a crash fault occurs when a process stops executing requests/steps. A process that crashes executes the algorithm correctly until some point in time *t*, when it stops operating and never recovers. Note that this model does not prevent a crashed machine from recovering, however it is assumed that after recovery, processes need to join the system as new processes.
- *Omission*: an omission fault occurs when a process does not send or receive a message that was supposed to be sent or received according to the algorithm. Omission faults are more general than crash faults as the faulty process may continue its execution after an omission occurs (unlike crash faults, where it is assumed that the process stops after the fault).
- *Crash with Recovery*: in this mode, a process is considered faulty if it either crashes and never recovers or is constantly crashing and recovering. A process that crashes and recovers a finite number of times during an execution is still considered to be correct. When a process recovers after a crash, it may need to update its internal state before interacting with other processes. Thus, techniques that make usage of a stable storage and logging mechanisms are deployed to help the recovery procedure. To ease the need of a stable storage we might consider that a set of processes never crash, so the requirement of having a stable storage is no longer required.
- *Eavesdropping:* this kind of faults may occur when an active adversary is present on the system. Leaks of private information exchanged by processes fall in this fault category. A faulty process

may leak information about its internal state and possibly give the attacker enough information to coordinate an attack. These faults can be prevented by applying secure cryptographic techniques to the private data.

- *Arbitrary*: faults that cause arbitrary deviations from the correct algorithmic behavior fall under this class. By making no restrictions on what can happen when a fault occurs, the arbitrary fault model is the most general. Arbitrary faults may have natural causes (for instance, electromagnetic interference), result from unintentional errors (a bug in a program), or be intentionally generated by a malicious attacker that was able to exploit a vulnerability of the application or of the operating system. Arbitrary faults are also often named Byzantine faults (Cachin, Guerraoui, and Rodrigues 2011; Lamport, Shostak, and Pease 1982).

From the faults hereby described, the focus of this dissertation is on systems that are able to tolerate arbitrary faults.

2.2 System Model

The types of faults are not the only factor that affect the resilience of the system. Other characteristics of the system, namely its behavior in the time domain, are also relevant. In fact, even relatively benign faults such as crash-stop may be hard to detect (and to tolerate) in a distributed system unless assumptions can be made regarding communication and processing delays in the system. In this context, it is important to distinguish the following relevant models of distributed systems: synchronous, asynchronous, and partially-synchronous systems.

- *Synchronous System*: a system is considered synchronous if there is a known upper bound on the processing delay (synchronous computation) and a known upper bound on the message transmission (synchronous communication). Since there is a known upper bound established on the delays of the system, it becomes much easier to detect failures: if a given process does not respond timely, it is considered to be faulty. However, building such a system requires an overall analysis of the complete infrastructure that composes the system such as hardware and software limitations; only then it is possible to conclude what might be a feasible upper bound, given the current limitations.

- *Asynchronous System*: a system is considered asynchronous if there is no timing assumptions on the processes and network links. Thus, in an asynchronous system there are no upper time bounds for communication and processing delays.

- *Partially-Synchronous System*: in practice, systems are designed and deployed such that they can respect pre-defined time bounds in normal operational conditions, i.e., they behave like synchronous systems most of the time. However, they are also not over-provisioned for the worst case. Thus, they may experience long delays under stress, for instance when the network or the CPU is overloaded due to a peak in the workload, i.e, they may behave transiently as an asynchronous system. These systems can be modeled by the property of *eventual synchrony*: a property that

states that the system will eventually behave as a synchronous system although there is no certainty of when this is going to happen. However it is not assumed that, when reaching synchrony, the system will stay synchronous or that its initial state is in an asynchronous period (Cachin, Guerraoui, and Rodrigues 2011). Many practical solutions for addressing Byzantine fault tolerance assume this model (Cachin, Guerraoui, and Rodrigues 2011; Kotla, Alvisi, Dahlin, Clement, and Wong 2007; Clement, Wong, Alvisi, Dahlin, and Marchetti 2009; Guerraoui, Knežević, Quéma, and Vukolić 2010).

2.3 BFT Concepts and Primitives

In this dissertation we are interested in studying protocols that can help to implement replicated services that tolerate Byzantine faults. As most of the literature on the subject, we will consider services that can be implemented using *state machine replication* (SMR). The SMR approach, originally proposed by Lamport (Lamport 1978b; Lamport, Shostak, and Pease 1982; Lamport 1978a) and adopted by most BFT approaches (Castro, Liskov, et al. 1999; Kotla, Alvisi, Dahlin, Clement, and Wong 2007; Abd-El-Malek, Ganger, Goodson, Reiter, and Wylie 2005; Cowling, Myers, Liskov, Rodrigues, and Shrira 2006; Clement, Wong, Alvisi, Dahlin, and Marchetti 2009; Bessani, Sousa, and Alchieri 2014) assumes that the service can be modeled as a state machine. The server starts with some initial state S_0 and then evolves by executing a sequence of commands c_1, c_2, ..., making a deterministic state transition in each step (i.e., the next state depends exclusively on the previous state and on the command being executed). As a result, if two correct replicas start with the same initial state, and execute the exact same sequence of commands, they end up in the same final state. Clients send commands to all replicas and receive back replies, and then apply some function to the set of replies received to mask server failures. If only crash failures can occur, $f + 1$ replicas are needed and the client can simply pick the first reply. If Byzantine failures can occur, at least $2f + 1$ replicas are needed, such that the client can always get a majority of correct replies (and discard the faulty replies, if any).

As described above, to ensure that correct replicas keep a consistent state, one needs to ensure that all correct replicas receive the same set of commands in the same order. Therefore, a protocol is needed to ensure that all commands are reliably broadcast to all replicas and totally ordered among each other. A protocol that achieves such goal is called an *atomic broadcast* protocol (Cachin, Guerraoui, and Rodrigues 2011). The designation “atomic” derives from the fact that it remains indivisible despite failures – the message is either delivered to all correct replicas or not, and, delivery appears to be “instantaneous” in time, resulting in a total order of all deliveries. Atomic broadcast can be implemented as a serial execution of several consensus instances where, in each consensus instance, one command is decided, i.e., replicas run consensus instance number one to decide the first command to execute, then they run consensus instance number two to decide the second command to execute, and so forth.

Moreover, the work by Dwork *et al.* (Dwork, Lynch, and Stockmeyer 1988) demonstrated that consensus can indeed be solved under a system that is partially-synchronous which represents the class of systems that this dissertation focuses on.

2.3.1 Byzantine Consensus

We now provide a more precise description of the properties that Byzantine Consensus satisfies. Distributed protocols can be characterized by safety and liveness properties, as described below:

- Safety: a safety property states that bad things do not happen (Cachin, Guerraoui, and Rodrigues 2011). In the consensus problem, a set of processes propose values and they have to decide on the on the same value. A safety property for consensus should state that the decided value is the same for all the processes and that the decided value was proposed by some correct process p.
- Liveness: a liveness property states that good events will eventually happen (Cachin, Guerraoui, and Rodrigues 2011). A liveness property for the consensus problem should state that the all processes will eventually decide on a value.

More precisely, Byzantine Consensus can be characterized by the following properties (Cachin, Guerraoui, and Rodrigues 2011):

- *Termination:* Every correct process eventually decides some value.
- *Strong validity:* If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or a special value $\perp$ (which means that no command is executed).
- *Integrity:* No correct process decides twice.
- *Agreement:* No two correct processes decide differently.

When used to implement atomic broadcast, the value v is a command issued by a client. Note that although a client only needs to receive $f + 1$ equal replies to make progress, in many scenarios it can be impossible to solve Byzantine Consensus with just $2f + 1$ replicas.

2.3.2 Views

As it will become clearer in the subsequent description, most protocols that implement Byzantine Consensus use a leader-based approach. Generally speaking, one of the replicas is elected as a leader and acts as a coordinator to facilitate the agreement process. If the leader is non-faulty, and is not suspected to be failed by the other non-faulty processes, it selects a client command and coordinates with other processes to ensure that such command is decided. If the leader stops, becomes extremely slow, or is Byzantine and does not comply with the specified protocol in a way that can prevent consensus from being reached, the remaining correct processes initiate a procedure to replace the leader, which is called *view change*. A view is, therefore, a list of processes that are participating in the consensus instances and their roles (which one is the leader). Since, to implement atomic broadcast, multiple instances of consensus need to be executed (one for each command), the same view is used in consecutive instances, until a reconfiguration is needed.

2.3.3 Recovery and Membership Changes

In a system that implements state machine replication, it is usually helpful to be able to add new replicas or to allow replicas that have stopped (either due to a crash or due to schedule maintenance) to later re-join the system. We have already discussed that most Byzantine SMR systems support a view change operation. This operation can be used not only to change the roles of the participating processes (i.e, to select a new leader) but also to add or remove replicas from the system. However, adding replicas to a running system introduces the additional problem of bringing the state of the replica up-to-date. This can be achieved by letting the new replica execute all the commands that have been executed by the active replicas, by copying the state from a correct replica, or by a combination of these approaches. For this purpose, important extensions like logging and checkpoint creation, state transfer and reconfiguration, greatly influence the protocol's execution.

- State Transfer: adding or removing replicas to a system can contribute to the extension of its lifetime while continuing to process clients' requests. The ideas of state transfer and reconfiguration are further explored in Section 2.5.3.1.
- Logging and Checkpoint Creation: logging is a technique used to store the actions performed by a given replica. When a reconfiguration to the system is being performed, some replicas may be in transient state trying to recover the current state. Logging the actions performed help the replica to reach that state. However, the log can grow to sizes that become computationally exhaustive to store or to perform a recovery on. Thus we can save a "snapshot" of the current process state to a reliable storage so that it can recover based on that information stored – this is called a checkpoint and it discards the log entries made until the taking of the "snapshot" thus reducing greatly the space requirements by the system. A recovering replica now only needs to ask for the last checkpoint in the system and build up the log from there.

2.4 BFT Protocols

One of the main barriers while deploying a BFT protocol is the high requirements needed to tolerate f faults – usually the number of servers n, must be greater or equal than $2f + 1$ or $3f + 1$ (this difference will be clear in this section).

With different protocols being more adequate for different workloads and environment conditions a system designer would need to choose the right technique under the assumption that while the system is running, the conditions will not deviate from the expected.

In this section we describe the Byzantine Generals Problem (Lamport, Shostak, and Pease 1982) which formalizes the problem of arbitrary faults and proposes two solutions while under synchrony. Then we describe three works that offer a solution to the problem described: PBFT (Castro, Liskov, et al. 1999), Zyzzyva (Kotla, Alvisi, Dahlin, Clement, and Wong 2007) and Aardvark (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009). PBFT was the first practical system that tolerated Byzantine faults under a partially-synchronous environment while Zyzzyva and Aardvark further explored variations of that system. Other variations such as HQ (Cowling, Myers, Liskov, Rodrigues, and Shrira 2006) and Q/U

(Abd-El-Malek, Ganger, Goodson, Reiter, and Wylie 2005) are not discussed. We chose Aardvark and Zyzzyva because they represent two protocols in completely opposites sides of the spectrum: Aardvark is more robust so it does not deviate much from its usual communication pattern when in the presence of faults while, on the other end of the spectrum we have Zyzzyva, a solution that offers high client throughput but is more fragile, i.e., in the presence of faults, it executes a fallback mechanism with an additional "recovery" step. These protocols should give insight of what the current state of the art is regarding solutions to the Byzantine problem under SMR.

The systems also assume that a strong adversary can coordinate faulty nodes in order to compromise the replicated service. However, the adversary cannot break cryptographic techniques (like collision-resistant hashes, encryption, signatures and authenticators). These systems use signed messages (symmetric or asymmetric ciphering) and/or Message Authentication Codes (MACs) for authentication purposes – usages between these methods are described when relevant for each protocol.

The protocols follow a leader based approach: within a given view, one replica is the leader (but it is still considered a replica which executes clients' requests as any other replica).

2.4.1 The Byzantine Generals Problem

Before beginning with the description of the protocols used to tolerate arbitrary faults, it is important to understand the underlying problem that those protocols try to solve. The description of the problem was formalized by Lamport in (Lamport, Shostak, and Pease 1982).

In 1978, Lamport L. depicted an algorithm that introduced the concept of SMR under a distributed environment (Lamport 1978b). From the concept of partial ordering, the algorithm described an extension that provided total ordering – the algorithm however produced arbitrary results if it was not coherent with the decision of the system's users. A solution for this arbitrary problem is by using synchronized clocks.

Additionally, the algorithm assumed that processors never failed and that all the messages were correctly delivered – it operated on a non-faulty system. Lamport then created a real-time algorithm (Lamport 1978a) that assumed upper bounds on message delays and that correct processes had their clocks synchronized. It is one of the first algorithms that described the idea of arbitrary faults described in The Byzantine Generals Problem (Lamport, Shostak, and Pease 1982).

The scenario used for explaining the Byzantine Generals Problem is an attack performed to an enemy city by the Byzantine army which has divisions commanded by their own general. Communication between generals are only possible via messenger. The objective of the Byzantine army is to reach a plan. However some generals are traitors and may disrupt consensus from happening.

From this point onwards, and in order to directly compare to other solutions, we use the term *leader* to describe a general, the term *replica* to describe a lieutenant, and a traitorous behavior means that the instance (leader or replica) is faulty. The leader is still considered a replica of the system (Replica 0).

The algorithm used by the instances must have the following guarantees:

- All correct replicas decide upon the same plan of action.

- A small number of faulty replicas cannot cause the correct instances to adopt a bad plan.

Every replica communicates its information to other replicas – $v(i)$ is the information communicated by the ith replica and by combining $v(1), ..., v(n)$ a replica can reach a plan of action. However, this solution does not work – a faulty replica may send different values to different replicas which means that correct replicas will have conflicting values. Therefore, two new requirements must be formalized:

- Any two correct replicas use the same value of $v(i)$.
- If the ith replica is loyal, then the value that he sends must be used by every correct replica as the value of $v(i)$.

Since the new requirements focuses on the the value sent by one replica (the leader), the problem can be reduced: given a set of n replicas, there is a single leader instance which sends orders to his $n-1$ replicas. The problem is now reduced to two *interactive consistency* conditions:

- **IC1**: All correct replicas obey the same order.
- **IC2**: If the leader is correct, then every correct replica obeys the order he sends.

We can also conclude that if the leader is correct then $IC2 \Rightarrow IC1$. The original problem can be viewed as an application of these conditions: each instance can be seen as a leader which sends his order $v(i)$ and all other instances act as replicas.

Now let's suppose we have a leader and two replicas (Replica 1 and Replica 2) and two possible decisions: attack or retreat. Additionally one of the three instances is faulty.

If Replica 2 is faulty, and the leader issues an attack order to Replica 1 and 2, then Replica 2 may also report to Replica 1 that the order given was to retreat. To satisfy IC2, Replica 1 should proceed with the attack order.

If the leader is faulty he may issue two different orders to Replica 1 and 2 respectively – by sending an attack order to Replica 1 and a retreat order to Replica 2, Replica 1 must obey the attack order since he does not know which instance is the faulty one. Similarly, Replica 2 will follow the retreat order thus violating IC1.

Concluding, no solution exists for three instances in the presence of one fault. From the result of (Pease, Shostak, and Lamport 1980) we need at least $3f + 1$ instances to tolerate f faults.

Symmetric Cypher Solution: A first solution with $3f + 1$ replicas uses symmetric cyphering when exchanging messages between replicas (it is described by the authors as the "Oral Messages Solution"). It admits the following properties:

1. Every message sent is delivered correctly.
2. The receiver of a message knows who sent it.
3. The absence of a message can be detected.

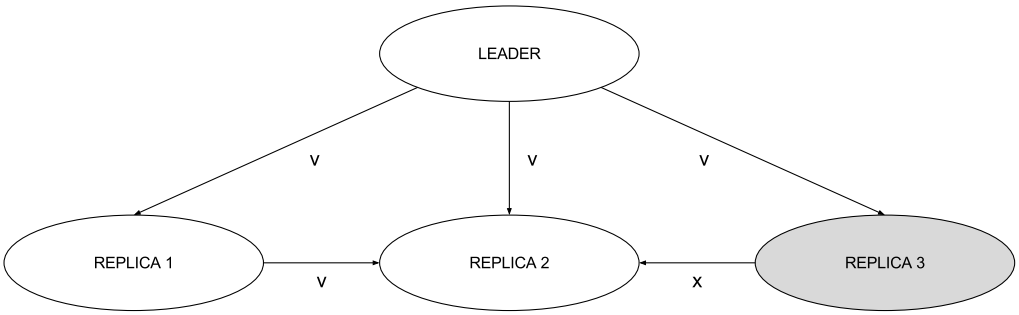


Figure 2.1: Replica 3 is faulty

Additionally the leader sends messages to all the other replicas. If the leader is faulty and decides not to send any message then the default action of the replicas that did not receive the message is to retreat.

The Symmetric Cypher algorithm, $SC(f)$, copes with f faulty instances and assumes a *majority* function. The steps are described in (Lamport, Shostak, and Pease 1982) but a quick summary of the algorithm is as follows:

1. The leader sends a command to every replica.
2. Each replica acts as a leader and sends out a command to all other replicas.
3. Each replica uses the majority function to compute the value based on the commands received in step 2.

A recursion of the algorithm occurs in step 2 – each replica acting as a leader will start the algorithm with $SC(f-1)$. So for $SC(f-k)$ the algorithm will be called $SC(n-1), ..., SC(n-k)$ times.

In Figure 2.1 we have the case where $f = 1$ and a replica is faulty. We can see that both Replica 1 and 2 receive the values $\{v, v, x\}$ therefore IC1 is respected because the majority of the set of responses is v. IC2 is also respected because all loyal replicas (1 and 2) respect the correct leader's order.

In Figure 2.2 we have the case where $f = 1$ and the leader is faulty. Despite the leader sending different orders to its replicas, all replicas arrive to the same set of orders $\{x, y, z\}$ so IC1 is respected (IC2 is not relevant since the leader is faulty).

As a remark, this is a simple example where we have only one fault. Remember that the algorithm is recursive so the communication is quite expensive in each step.

Asymmetric Cypher Solution: In the Symmetric Cypher Solution, a replica could lie about the leader's order: a replica that receives a given message only knew that it was sent from a given replica (thus not knowing if the message was modified). If the ability to modify a message is restricted, the problem becomes easier – by signing messages, we can add two additional assumptions to the messages interchanged between instances:

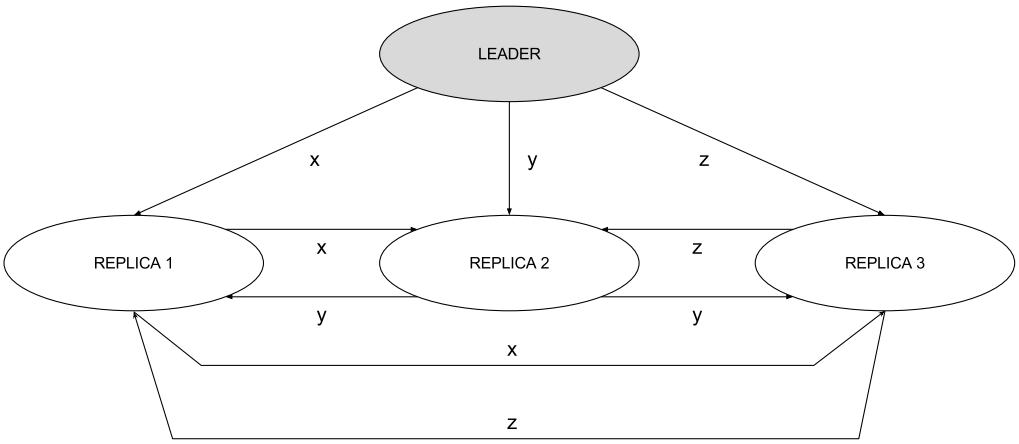


Figure 2.2: Leader is faulty

1. A correct leader's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
2. Anyone can verify the authenticity of a leader's signature.

The Asymmetric Cypher algorithm, $AC(f)$, proposed in (Lamport, Shostak, and Pease 1982) tolerates f faults but the requirement of at least 4 instances no longer holds – the problem can be solved with 3 instances given $f + 2$ for the number of total instances. In this algorithm, each replica maintains a set V that stores received orders that were properly signed. A new function, $choice(V)$, is also assumed by the algorithm and it needs to have the following requirements:

1. $choice(V) = v$ if V contains only one element.
2. if V is an empty set then $choice(V) = RETREAT$.

The algorithm starts with the leader sending a signed order to all the other replicas. When a replica receives an order (either by a leader or other replicas), it verifies its signature and, if valid, puts it in V. If the replica has received k signed messages and $k < f$, then he sends v to all other replicas that might have not seen v. When Replica i does not receive any more messages, he obeys the order given by $choice(V_i)$.

All loyal replicas will eventually compute the same set V and by using a deterministic function $choice(V)$, IC1 is respected. Also if the leader is loyal, IC2 is also respected (all correct replicas will have the same V).

In Figure 2.3 is a demonstration of an execution of $AC(1)$ where the leader is faulty. The message *"attack":0* means that the order *attack* was signed by replica 0 (leader). Furthermore, *"attack":0:1* means that the previous message was additionally signed by Replica 1. In this situation, the leader issued an "attack" order to Replica 1 and a "retreat" order to Replica 2. After the communication step, each Replica arrived to the same set where $V_1 = V_2 = \{"attack", "retreat"\}$. Each Replica arrives to the conclusion that the leader is faulty since he signed two different orders and the signature cannot be forged.

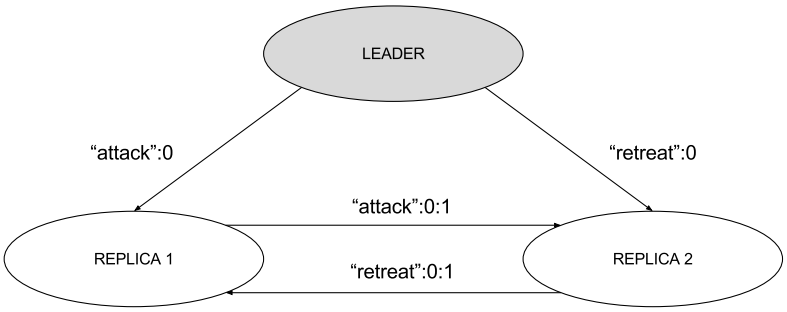


Figure 2.3: Leader is faulty

An important remark must be made about the assumptions made on the messages' transmissions – a faulty instance can delay or not transmit at all a given message but, this delay or message absence can be detected. To detect this failure there are two timing assumptions being made (which are dominant in synchronous systems):

1. There is a fixed maximum time needed for the generation and transmission of a message.
2. The sender and receiver have clocks that are synchronized to within some fixed maximum error.

Despite the formalization of two solutions in this section (one requiring $3f+1$ replicas and the other requiring $2f+1$ replicas), these solutions were designed for synchronous systems so we cannot assume that they work on an asynchronous environment like the Internet.

The following algorithms explained in this dissertation (Sections 2.4.2, 2.4.3, 2.4.4) explore solutions under a partially-synchronous system. In these systems we can assure both *liveness* and *safety* properties when we have synchrony but, only *safety* can be guaranteed in periods of asynchrony (Fischer, Lynch, and Paterson 1985). They are still based in SMR, require $3f+1$ replicas and there is a finite number of clients where any number of which can be faulty. To expand on the requirement for the number of replicas needed in these types of systems where synchrony is not guaranteed, there are two relevant properties that need to be stated:

- *Intersection*: any two quorums have at least one correct replica in common.
- *Availability*: there is always a quorum available with no faulty replicas.

In order to provide liveness we need to assume that we will only receive $n-f$ responses (f replicas may not respond) – this is the quorum size ($=2f+1$). To provide correctness, two quorums must intersect at least in one correct replica: $(n-f)+(n-f)-n \geq f+1$ thus the result $n \geq 3f+1$.

2.4.2 PBFT

Castro and Liskov, introduced the first SMR BFT protocol that is safe under partially-synchronous systems – PBFT (Castro, Liskov, et al. 1999).

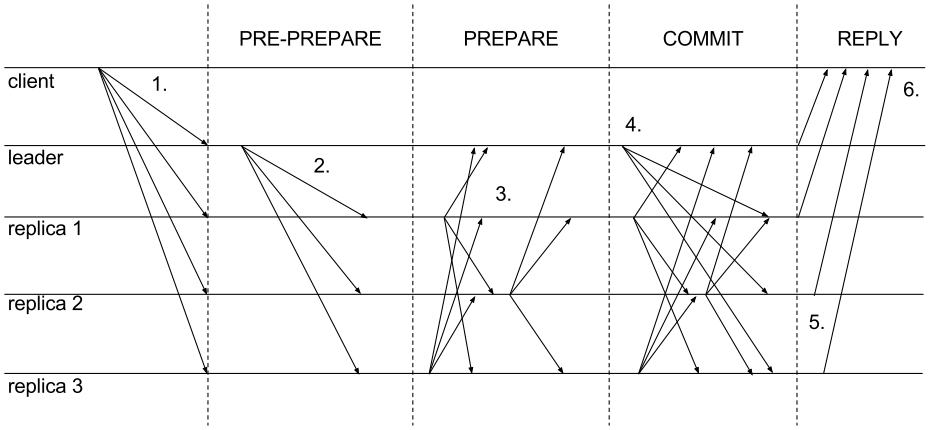


Figure 2.4: PBFT Communication Pattern

PBFT uses a three-phase commit communication pattern: PRE-PREPARE, PREPARE, and COMMIT. The messages are authenticated using MACs. PRE-PREPARE and PREPARE are phases that totally order the requests for a given view (even when the leader, which is responsible for the ordering, is faulty). Each replica keeps a message log to register the messages that it has received or sent. The normal execution of the protocol is represented in Figure 2.4 and the steps performed are as described below:

1. Client sends request to a replica – a client sends a request to what it believes to be the leader. If it doesn't receive a response within a time bound then it retransmits the request to all other replicas. The replica then verifies the message by checking if the client's request is within the load limits of the server. If a replica, other than the leader, receives the client request, it authenticates the message and sends it to the leader.

2. Leader sends a PRE-PREPARE message to all Replicas – When the leader receives a client request, it assigns a sequence number n and registers the message in its log. Then it multicasts a PRE-PREPARE message to all other replicas containing the message and the current view number v. A replica only accepts a message if it is in the same view and its successful in the verification of its authenticity.

3. Replica receives PRE-PREPARE from the leader and sends PREPARE to all other replicas – upon receiving the PRE-PREPARE message, the replica authenticates the message. If the replica has already accepted the message then it is discarded. If the replica has already processed the message (same sequence number) in the current view, then it is discarded. The replica also checks for the integrity of the MAC authenticator used in the message: if it is invalid, the message is discarded. If the authenticator is valid then the replica registers the PRE-PREPARE message and sends a PREPARE message to all other replicas along with a digest of the requests present in the PRE-PREPARE message.

4. Replicas then collect a quorum of $2f$ matching PREPARE responses proving that a quorum has agreed to assign the sequence number n to the message in view v. However, this is not sufficient to guarantee that each correct replica has the same sequence of messages as another correct replica in the same view – a replica might have received the prepare message in a different view but

the request has the same sequence number. That is why there is a commit phase – each replica multicasts a COMMIT message stating that it has indeed received a quorum of prepared messages (the commit message is added to the log). When a replica receives $2f$ commit messages it already has the needed quorum since the replica itself is prepared to commit the request. The replica only executes the request after executing any pending requests with lower sequence numbers.

5. Replica receives $2f + 1$ COMMIT messages and sends a REPLY to the client – after receiving $2f + 1$ matching responses (from distinct replicas), the replica executes the request, and sends a REPLY to the client containing the result of the execution.

6. The client then waits to receive $f + 1$ messages with valid MACs and with the same result and time in which the request has been made and concludes the execution.

2.4.3 Zyzzyva

Zyzzyva (Kotla, Alvisi, Dahlin, Clement, and Wong 2007) is a BFT-protocol that uses speculation in order to reduce the cost of the BFT replication. In this protocol, replicas reply to a client's request without running an expensive consensus protocol. Instead, replicas rely only on the order proposed by the leader, then process the ordered requests and immediately respond to the client. The client however has an additional task: if the client detects any inconsistencies with the responses received, it helps in the convergence of the responses by the correct replicas. By using speculative execution, Zyzzyva can achieve high request throughputs.

The protocol relaxes the condition: *a correct server only emits replies that are stable*. Instead, realizing that this condition is stronger than needed, Zyzzyva's core idea is based on the role of the client in the system: *a correct client only acts on replies that are stable*. This weaker condition avoids the all-to-all communication between replicas to reach consensus.

Ultimately, the challenge is ensuring that responses to correct clients become stable. While this task is given to the replicas, a correct client can greatly speed the process by supplying information that would make the request become stable or even lead to the election a new leader.

Zyzzyva's SMR protocol is executed by $3f + 1$ replicas and it is based on three subprotocols: agreement, view change and checkpoint. The *agreement* protocol executes within a sequence of views where a replica(*leader*) leads the agreement subprotocol. The *view change* subprotocol is responsible for the election of a new leader (due to the current faulty one). The *checkpoint* subprotocol reduces the state that is stored within each replica thus optimizing the *view change* subprotocol.

A request completes at a client when the client has a sufficient number of matching responses ensuring that all correct replicas will execute the request. The client can determine when the request completes since the client receives responses from replicas that include the reply to the application and the history. The history contains all requests executed by the replica prior to (but including) the current request.

There are three cases that are considered for Zyzzyva's agreement protocol: the *fast case*, the *two-phase case* and the *view change* case.

In Zyzzyva's *fast case*:

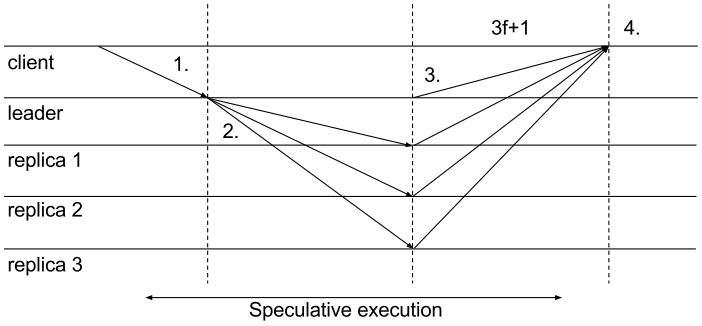


Figure 2.5: Zyzzyva's *fast case* Communication Pattern

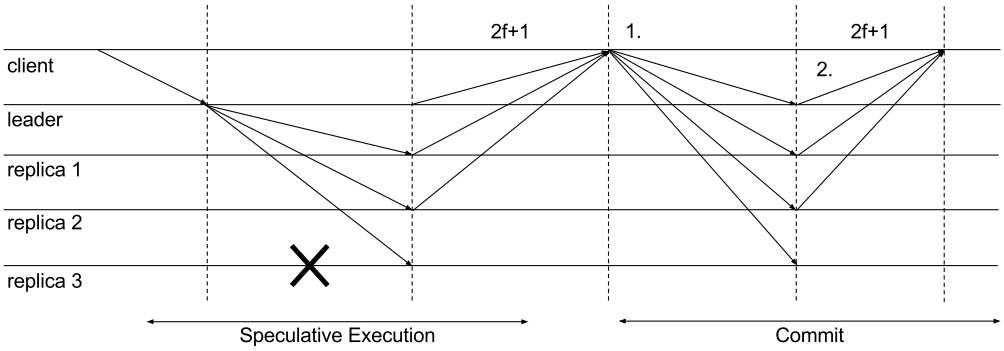


Figure 2.6: Zyzzyva's *two-phase* Communication Pattern

1. The client sends a request to the leader. The request sent by the client to the leader has a timestamp that guarantees only one execution of the request (exactly-once semantics).

2. Upon receiving the request, the leader assigns a sequence number and adds a cryptographic one-way function (hash) for the request sent by the client. The leader then forwards the ordered request to all the other replicas.

3. The replica receives the ordered request from the leader and, optimistically, assumes that the leader is correct. It then adds the request to its history, speculatively executes it and responds to the client. Additionally, the replica only executes the request if the message from the client is well-formed, the digest matches the cryptographic hash of the message and the leader forwarded the message in the same view where the replicas receive it.

4. The client receives $3f + 1$ matching responses and completes the request. Without any faults or timeouts the $3f + 1$ responses will match and the client can safely rely on the request result.

Zyzzyva's *two-phase* communication pattern happens if some of the replicas are faulty or slow. It applies when the client receives between $2f + 1$ and $3f$ responses. All the communication is exactly as it was described in the *fast case* except for the client's reception of the responses.

1. The client now receives between $2f + 1$ and $3f$ responses, assembles a *commit certificate* and relays it to all the replicas. When the client sends the request, it sets a timer. After the timer expires, if the client gathered between $2f + 1$ and $3f$ matching responses then it has proof that

the majority of the replicas agree on the ordering in which the request should be processed. The replicas however, do not know that such agreement quorum exists – they only know about their local execution.

A problem that may arise at this point is when a *view change* occurs. Since the view $v+1$ must be consistent with the state in view v the *view change* subprotocol must know which requests were executed in view v. Since up to f replicas may be faulty the *view change* subprotocol must use information from a quorum of $2f+1$ replicas.

To solve this problem the client sends a message containing a *commit certificate* that contains a list of $2f+1$ replica signed-portions of the response given to the client and the corresponding $2f+1$ replica signatures.

Replicas then receive the message from the client containing the *commit certificate* and acknowledge its reception to the client. A replica upon receiving the *commit certificate*, can validate if the request should be executed in the current view (since the request has its own sequence number and history). Note that the request was already executed by a correct replica so it shouldn't be executed again to respect exactly-once semantics.

2. The client then only needs to receive the confirmation by $2f+1$ replicas to consider the execution to be completed. Since the *view change* subprotocol needs $2f+1$ replicas for execution we know that at least $f+1$ replicas are definitely correct and stored the commit certificate.

The *view change* case occurs when the client receives fewer than $2f+1$ responses or the client suspects a faulty leader.

When the client receives fewer than $2f+1$ matching responses it resends the request message to the replicas (so they can track the progress and possibly initiate a view change). Upon receiving the request from the client, a replica checks if the request has a higher timestamp than the cached response for that client and then sends a message to the leader while initiating a timer. If the replica receives a response from the leader then it proceeds with the protocol as described earlier. If the time runs out then the replica starts a view change. To maintain exactly-one semantics, all replicas have a cache of the last reply for each client.

If the client receives valid response messages from the same request but with different sequence numbers or history, then it is declared as a Proof of Misbehavior (Aiyer, Alvisi, Clement, Dahlin, Martin, and Porth 2005) on the leader. Noticing this, the client sends a message to all replicas with the Proof of Misbehavior and a view change will initiate.

2.4.4 Aardvark

Since PBFT, many systems have appeared that promised high throughputs under Byzantine conditions (Q/U, HQ and Zyzzyva). However the high throughputs were often reached by relaxing some conditions – Zyzzyva moved many responsibilities to the client for the recovery of a system whenever a fault was detected. Relaxing such conditions can render a system unavailable for long periods of time – the system is *fragile* to faults.

The basic idea of Aardvark (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009) is to build a BFT system that offers high throughput while maintaining robustness so that the system offers predictable performance (even under faults). Aardvark however deviates from some of already established "conventional wisdom" – it uses signatures instead of MACs, despite being considered an important performance bottleneck (Castro, Liskov, et al. 1999), performs regular view changes even when they may disrupt the service temporarily and the use of point to point communication instead of IP-multicast.

The authors define two important concepts that describe the execution conditions:

- Gracious Execution: an execution is considered gracious if it is synchronous (known short timeout on a message delay) and all clients and servers are correct.
- Uncivil Execution: an execution is considered uncivil if it is synchronous (known short timeout on a message delay) and up to *f* clients and servers are Byzantine.

Aardvark rejects any optimizations while the system is under a *Gracious Execution* that may degrade the performance of the system while in *Uncivil Execution*. The system also operates under an asynchronous network where *Synchronous Intervals* occur. *Synchronous Interval* is a time interval where any message sent by any correct process (client or server) is delivered within a defined time bound.

The main contributions of Aardvark are also the three main core differences between previous BFT systems: signed client requests, resource isolation and regular view changes.

- The usage of signed client requests comes from the property of non-repudiation and ensures that all correct replicas validate the client request identically thus "eliminating a number of expansive and tricky corner cases found in existing protocols that make use of a weaker (though faster) message authentication code (MAC) authenticators" (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009) – "one node validating a MAC authenticator does not guarantee that any other nodes will validate that same authenticator". While signed requests are expensive, they are only used to sign client requests and this is done on the client side (avoiding this expense on the server side). Since signing requests cryptographically is an expensive operation, an attacker could send a large number of requests that need to be verified on the server side. To limit this, Aardvark puts a hard limit on the number of incorrect/faulty signatures that a client can send to the system (by using a hybrid MAC-signature) and it also forces the client to complete one request before sending another one.
- For resource isolation, Aardvark deploys unique Network Interface Controllers (NICs) – each replica has a one-to-one channel that is used for message exchange. This allows each replica to defend itself against attacks (by disabling the attack source NIC). It also prevents a faulty replica to interfere on the message delivery by correct replicas. However, by having unique interfaces for communicating between pairs of replicas, the system suffers from a performance hit – it does not use multicast to optimize all-to-all communication. To optimize the usage of each interface, Aardvark differentiates client requests with the communication between individual replicas by using different work queues. By doing this, Aardvark prevents client requests to interfere on replica-to-replica communications.

- Aardvark also changes views regularly. Replicas monitor the performance of the current leader and continuously raise the acceptable throughput level. When the leader fails to correspond to the current level, replicas start a view change. View changing was treated as an expensive protocol and only used when the throughput was dropping fast or other extreme situations. However the view change protocol "is similar to the regular cost of agreement". While doing a view change, the system cannot process new requests but performing a view change only when we have a faulty leader is more costly than regularly changing the view (Clement, Wong, Alvisi, Dahlin, and Marchetti 2009).

The steps performed by the protocol follow a standard three phase commit protocol (Castro, Liskov, et al. 1999) that was already explained in Section 2.4.2. However, the steps are performed using the three main core differences already explained.

As described, replicas frequently monitor the performance of the leader: by slowing increasing the throughput rate which the leader must satisfy. Moreover, Aardvark expects from the leader a frequent supply of PRE-PREPARE messages and high, sustainable throughput. A timer is set and whenever a PRE-PREARE message is issued the timer resets. If the timer expires, a view change is initiated. There are also periodic checkpoints where replicas assess the throughput of the system. If the performance drops below a certain threshold then a view change is initiated.

Another issue is that a faulty leader, to avoid being replaced, may issue many PRE-PREPARE messages. However it will be caught on the periodic checkpoint where replicas will check the current throughput of the system (completed requests) and, therefore, the leader will face a demotion.

2.5 BFT Adaptation

As we have described in the previous section, BFT protocols are relatively expensive and require the exchange of a significant amount of messages during their execution. Also, their performance depends on operational context: some protocols are more efficient in stable runs while others offer better performance under attack. Furthermore, the number of messages exchanged depends on the number of processes that exist in the system (which is often a function of the number of faults that needs to be tolerated). Finally, as in any fault-tolerant protocol, nodes may fail and need to be replaced. Since many of the factors that affect the performance of a BFT protocol may be hard to estimate *a priori*, and may change dynamically, it is relevant to consider the design and implementation of BFT protocols that can adapt in run-time.

When considering the design of adaptive BFT protocols, there are three different aspects that need to be considered:

- *What to adapt,* i.e., what are the possible system configurations that may be selected. In this dissertation we mainly consider four different adaptations: integration of new replicas (or re-integration of replicas that have recovered) to replace failed replicas, dynamic changes to the number of replicas, migration of replicas between virtual containers to avoid recently uncovered vulnerabilities on the hosting machines, and dynamic changes to the BFT protocol itself.

- *When and why to adapt,* i.e., what are the policies that drive the selection of the target configuration of the system given a system state. In this aspect there are two facets to consider. One is where the adaptation code is executed: it can be embedded in the BFT protocol (what we call a *monolithic approach*) or it can be implemented in a separate module that is in charge for the reconfiguration (what we call a *modular approach*). To achieve modularity, a BFT implementation may provide an interface exposing methods to allow its reconfiguration, thus separating the reconfiguration mechanisms from the rules that trigger that same reconfiguration (Lamport, Malkhi, and Zhou 2010). The other aspect is to distinguish implementations where the system is only able to execute a fixed set of policies, that are hard coded in the implementation from implementations that are flexible enough to implement a variety of policies. Typically, modular implementations are more amenable to implement a richer set of policies.
- *How to adapt,* i.e., what algorithms are used to perform the system reconfiguration. While, in principle, it is possible to reconfigure a system by first halting its operation and restarting it under a new configuration, such approach is not very appealing given that it may create long periods of service unavailability. Therefore, reconfiguration algorithms attempt to support the dynamic reconfiguration with minimal impact on the performance of the system.

In the following paragraphs, we address each of these aspects in more detail.

2.5.1 What to Adapt

In this dissertation, we focus on the following set of adaptations that can be applied to a BFT implementation:

1. *Replica reintegration*: As any other fault-tolerant protocol, a BFT deployment can only tolerate a fixed number of faults at a given time. To ensure that the system will keep operating after failures occur, it is therefore fundamental to replace replicas that have failed, by correct replicas. The new replicas may be completely fresh or may already contain some some state that has been collected by that replica or by some other replica before the crash. In any case, the new replica will never be completely up-to-date (with regard to the state of active replicas) and some re-integration protocol must be executed to put its state on par with the current active correct replicas. Also, we have seen that many BFT implementations keep a *view* of what replicas are correct; therefore, replica reintegration typically involves the execution of the view change sub-protocol.
2. *Changing the number and/or location of replicas*: The replica re-integration process described above can be seen as a particular case of a more general adaptation that consists in the process of adding, removing or migrating replicas in a BFT deployment. Changing the number of replicas in runtime allows to scale the resilience of the system according to the estimated level of threat. In particular, the number of replicas may be increased if the risks of having machines compromised is higher and lowered when the risks decrease. Adjusting the number of replicas is important because, as we have seen, in BFT protocols there is an inherent trade-off between the resilience and the performance of the deployment. Replica migration can be helpful if a machine (where a replica is running) lacks resources or has some vulnerability that has been exposed.

3. *Changing the BFT protocol*: As we have seen, different BFT protocols exist and none of them offers the best performance in all scenarios. On the contrary, some perform better in stable scenarios while others perform better when the system is unstable. Since in a real deployment the environmental conditions may change in runtime, it is interesting to support the dynamic adaptation of the BFT protocol in use, in order to execute the protocol that is more suitable for the observed conditions. The biggest challenge in changing the BFT protocol is to coordinate the replicas such that the properties of the service are guaranteed during the change. The most straightforward manner of achieving this is to stop one protocol at all replicas before activating the execution of the new protocol. This may not be trivial, because one needs to assure that different protocols do not make inconsistent decisions about the same message. Therefore, existing BFT protocols need to be adapted/extended in order to support the concept of *abortability* and correctly maintain correctness when transitioning from *State A* to *State B*.

2.5.2 When and Why to Adapt

In this dissertation we are interested in building a modular solution to drive the adaptation. Therefore, instead of supporting a fixed set of policies and protocols that are hardcoded in a monolithic BFT implementation, we aim at a solution where there is a separate adaptation manager that can use a number of actuators to drive the reconfiguration of the BFT implementation. The adaptation manager can be seen as a logically centralized component that monitors the system operation, extracting metrics regarding the observed performance (bandwidth, memory usage, number of clients, etc...) but also information about system vulnerabilities and level of threat (for instance, resorting to intrusion detection systems). Using this information, the adaptation manager uses adaptation policies to select if and when the system must be reconfigured. The choice of the correct adaptation can be driven by user-specified rules (for instance event-condition-action rules (Guerraoui, Knežević, Quéma, and Vukolić 2010)).

We now make a brief overview of some of the policies that have been proposed in the literature, for the adaptations listed in the previous section:

- Castro and Liskov (Castro and Liskov 2000) suggest that replica re-integration should be performed not only to replace replicas that have failed but also as a *proactive* measure of defense against replicas that may have been compromised, even when they remain asymptomatic. The idea is that, in most cases, it takes a reasonable amount of time for an adversary to compromise a machine. Therefore, if the machine reboots and a fresh copy is re-installed, this may foil the efforts of an attacker and effectively prevent the adversary from taking more that $1/3$ of the replicas. This policy became known as *proactive recovery*.
- The BFT-SMaRt system (Bessani, Sousa, and Alchieri 2014) has a module for reconfiguration which allows replicas to be added or removed from the system. The paper also describes the trade-offs involved in this reconfiguration. With more replicas, the system can tolerate more arbitrary faults (at the cost of more messages going through the network). Since adding replicas has a cost associated with it (state transfer and view change), we should only add replicas when strictly needed.

- Abstract (Guerraoui, Knežević, Quéma, and Vukolić 2010) is a system that provides interchangeability between different BFT protocols. The paper focuses on the mechanisms required to replace the BFT implementation but also suggests some simple policies to trigger a protocol switch. These policies are static and hard-coded in the BFT implementations. The paper proposes to switch from a protocol that has good performance with favorable scenarios to a more expensive (but also more robust) protocol, when the leader is unstable. Switching happens when the leader is suspected of being malicious/faulty and, consequentially, the system remains in the more expensive configuration for a quarantine period before it reverts back to the optimistic protocol.

- Adapt (Bahsoun, Guerraoui, and Shoker 2015) further explores this idea by introducing an evaluation step for electing the next active BFT protocol. Adapt is an adaptive abortable BFT system that reacts to changing conditions of the environment while introducing Machine Learning techniques for the protocol selection process. Adapt uses an adaptation manager that has three operating modes: *static*, *dynamic*, and *heuristic*. The *static* mode only analyses the default characteristics of each protocol (we can say that they are the static features that represent the protocol such as the number of replicas needed to tolerate *f* faults). The evaluation process in *static* mode is performed before the system starts: the protocol chosen in this this step will be the starting protocol. In *dynamic* and *heuristic* modes, the evaluation process is done in run-time while evaluating the performance of the active protocol under the conditions in which the system is present. Moreover, the *heuristic* mode uses heuristic rules for the evaluation step. Adapt also uses two performance metrics for the evaluation process. The *Key Characteristic Indicators* (KCI) represent the static features of a given protocol (toleration of client faults, number of replicas, etc...). The *Key Performance Indicators* (KPI) are metrics that are dynamically computed (using prediction methods) and represent the performance of the protocol in run-time (such as progress and throughput). The KPI metrics are computed experimentally and obtained using techniques like Support Vector Machines or Regression (Bahsoun, Guerraoui, and Shoker 2015) – each protocol runs a period of time, while the *impact factors* change in order to get the KPI values under each state. After collecting a set of KPI values, we have a training set that is used to train a *prediction function*. The *prediction function* takes as input the *impact factors* and outputs the corresponding KPI value. In order to improve the *prediction function*, the training set is also updated while the Event System (ES) sends new events allowing the function to tune itself.

One important factor that needs to be taken into account is the cost of protocol switching since it has an associated overhead. After the selection of a new configuration by the adaptation manager, we know that it will be the best protocol present in the library given the new conditions. But if the benefits of switching are not significant to the performance, then the system would suffer the cost of protocol transition without gaining too much from that same cost. Therefore, the Adapt (Bahsoun, Guerraoui, and Shoker 2015) system uses a *switching threshold*, defined as follows:

$$\frac{p_{max}}{p_{curr}} \geq S_{thr}$$

Where p_{max} is the score for the best protocol chosen by the evaluation process, p_{curr} is the score for the currently active protocol and S_{thr} is the threshold value that can be defined by administrators.

Note that the adaptation manager itself can be a target of malicious attacks so some solutions re-

quire that the reconfiguration module is also replicated among the existent replicas. Thus, the replication adaptation manager will also need to execute an agreement protocol to decide on the reconfiguration result. Given that the adaptation manager only needs to agree on reconfigurations sporadically, the performance of the BFT protocol used to replicate the adaptation manager is not as critical as the performance of the BFT protocol supporting the application. Therefore, any robust BFT protocol may be used to coordinate the replicas of the adaptation manager.

2.5.3 How to Adapt

We now discuss some of the challenges involved in performing the adaptations listed in Section 2.5.1.

2.5.3.1 Replica integration.

One of the most heavy-weight tasks associated with adding or re-integrating a replica is the process of bringing the replica up-to-date. Typically this is done by transferring state from other active replicas. Several strategies have been proposed to increase the efficiency of this process, including:

- *Incremental Transfer*: hierarchical state transfer and incremental cryptography (Castro and Liskov 2000).
- *Collaborative State Transfer*: the state transfer is performed in a collaborative way with each replica contributing to the replica recovery (Bessani, Santos, Felix, Neves, and Correia 2013).

2.5.3.2 Changing the underlying protocol.

Abstract (Guerraoui, Knežević, Quéma, and Vukolić 2010) is an adaptive BFT system that proposes a protocol that implements a safe transition between two distinct BFT implementations. Abstract is based on the concept of *instances*. Instances are like a replicated state machine service and operate like so (they have a set of replicas that receive the clients' requests, execute them and return the result to the client). Each reply to the client contains a commit history which has a totally ordered sequence of client requests. An important concept introduced in Abstract is the concept of *abortability* – if some *progress* conditions are not met while executing a request, the request is aborted. These *progress* conditions are determined by the system administrator and the task of setting what is considered progress is facilitated: the administrator only needs to establish in which conditions the system makes progress instead of designing a system that tries to make progress under all conditions. Abstract allows instance reconfiguration like traditional state machine reconfiguration approaches (Lamport, Malkhi, and Zhou 2010) but the difference is that the reconfiguration is performed on top of *abortable* state machines. Each instance has a unique identifier (instance number), a number of replicas that implement the service and the protocol being used (along with other internals such as the current view number, leader, etc...).

The reconfiguration protocol of Abstract can be described in three different steps:

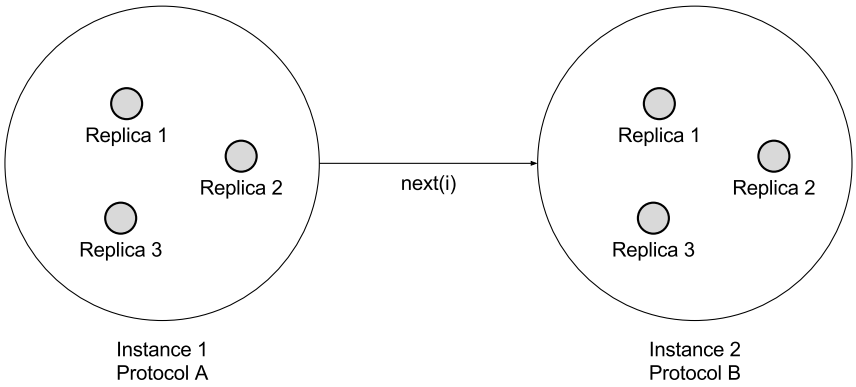


Figure 2.7: Abstract with two configured instances

1. Stopping the current Abstract instance: an abstract instance is immediately stopped as soon as it aborts a single request. Along with the abort signal, the instance also returns an *abort history* that contains a mapping between the replica's identification number *i* and its commit history.
2. Choosing the next Abstract instance: along with the abort signal, the aborting Abstract instance i also returns the identifier of the next instance $next(i)$. As with consensus, the result for $next(i)$ should be the same across all abort indications of instance i. Other property is that $next(i) > i$.
3. Combining the commit histories: the client uses the abort history of instance *i* in the invocation of $next(i)$. This will be referenced as the *init* history of the instance $next(i)$. The *init* histories are used to initialize an instance before it starts its own execution (accepting and executing clients' requests).

Abstract also provides three important properties:

1. Instance switching is idempotent: the composition of two Abstract instances yields another Abstract instance.
2. A correct implementation of an Abstract instance always preserves the safety conditions of a state machine (such as the total order of committed requests).
3. A replicated state machine is an Abstract instance that never aborts.

One important rule is required while running Abstract – there is only one active instance in the system and that instance is the only one that can commit requests. As with many other system models, Abstract assumes that links between processes are asynchronous (with periods of synchrony) and unreliable, and the total number of replicas present in the system is $3f + 1$ where up to f processes can fail arbitrarily (Byzantine processes). A strong adversary may coordinate faulty nodes but cannot break the cryptographic assumptions of hashing, authentication codes and signatures. Abstract also assumes that synchronous periods occur allowing the processes to communicate in a timely fashion (there is a known upper bound *t* for message transmission by correct processes). With this model, an administrator can pick different Abstract instances (each one representing a different protocol) and decide in which conditions each instance runs.

In the Abstract paper, these mechanisms are illustrated by performing dynamic adaptations between two BFT protocols, one derived from Zyzzyva (Kotla, Alvisi, Dahlin, Clement, and Wong 2007), namely ZLight that captures Zyzzyva's fast case and another called Backup that is based on PBFT. The Backup instance only works on top of a valid init history (one produced by a valid ZLight instance) – upon receiving a valid commit history, the instance state is reconstructed by executing every request provided by the history. After the state restoration, Backup commits exactly k requests and, after the execution of the kth request, it aborts and sends its commit history (digitally signed) back to the client. The choice for the value of k is up to the system administrators but it is important to know that it must be a value that contributes to the progress of the system – it should allow the Backup instance to be active for a long period of time (since we are in a state to handle failures) but also short enough so that it does not confine clients to the Backup instance for too long (since ZLight is capable of doing more progress under the "common case" conditions).

Adapt (Bahsoun, Guerraoui, and Shoker 2015) further explores the ideas above by introducing an evaluation step for electing the next active BFT protocol. Adapt is an adaptive abortable BFT system that reacts to changing conditions of the environment while introducing Machine Learning techniques for the protocol election process. Adapt offers three main benefits over an Abstract implementation:

1. No protocol order has to be defined a priori: the way that Abstract changes between instances needs to be already known before deploying the system. In AZyzzyva, the system was designed to change between the ZLight instance and the Backup instance – the two instances were known beforehand as the conditions that would trigger the switch. In Adapt we are not constrained by the number of protocols or by which order the instances change.

2. Switching can occur when we can gain performance or increase progress: Adapt is not constrained by a set of conditions in which the protocol change only occurs when there is a failure – if a performance boost can be achieved by switching the active protocol, then that change will be applied.

3. Fallback mechanisms are no longer needed – AZyzzyva has Backup instance as a fallback mechanism to guarantee that progress is being made while the "common case" conditions are not met. In Adapt the change occurs when there is a more appropriate protocol for the new conditions and no fallback mechanism is needed if those conditions maintain.

Summary

In this chapter we described the related work that we consider relevant for this dissertation. We started by describing the fault model of the systems analyzed and described the concepts and primitives related to Byzantine fault tolerance. We then described the work of Lamport on state machines that formalized the problem of arbitrary faults thus initiating the study of developing practical solution that tolerated Byzantine faults. The first practical solution, PBFT, started a growth in the development of these protocols – we have described Aardvark and Zyzzyva, two distinct protocols that formulate different assumptions for their execution. Finally we described techniques that performed adaptations on BFT

systems – we have introduced the concept of abortability and the mechanisms that are inherent to the adaptations being performed.

The next chapter will introduce the architecture and implementation details of our system.

3 ByTAM

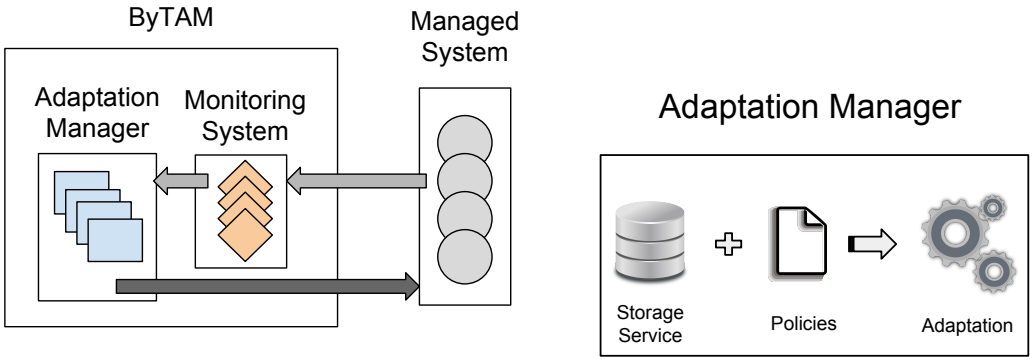
In this section we present the system model, the architecture, and a general perspective on how the different components of ByTAM operate to perform adaptations in a coherent way.

3.1 System Model

We assume a Byzantine failure model on which processes that fail can behave in an arbitrarily manner (Lamport, Shostak, and Pease 1982) and the existence of a strong adversary, with the ability to coordinate failed processes to compromise the replicated system. However, we assume that this adversary cannot violate known cryptographic techniques such as MACs, encryption, and digital signatures. Furthermore, we assume that at most f replicas of each component may fail (from either the adaptation manager, sensors or the managed system). Finally we assume an asynchronous network where eventually synchronous intervals occur; in those intervals, messages are delivered in time and the protocol makes progress.

3.2 Architecture

ByTAM has two subsystems which operate independently: the Monitoring System and the Adaptation Manager. Both communicate with the Managed System, as it is shown in Figure 3.1(a). The Managed System is the service that is provided to the end user; it provides information on the operational conditions and receives adaptations, working as a client of ByTAM. An example of a possible managed system is DepSpace (Bessani, Alchieri, Correia, and Fraga 2008), a fault tolerant tuple space. The Managed System must satisfy two properties: it must have some kind of reconfiguration mechanism (eg. view change protocol or support *Abortability*) and be monitorable (exporting performance and operational metrics). The Managed System operates independently from the remaining system components so, faults that prevent communication with ByTAM (network partitioning, asynchrony) may cause delays in the adaptation process. Additionally, the Managed System can have a different failure model from ByTAM. However, to achieve a globally robust solution, in this dissertation the Managed System is also Byzantine fault tolerant. On the following sections, we will describe the Monitoring System and the Adaptation Manager, presenting their main functionalities.



(a) System Architecture

(b) Adaptation Manager Services

Figure 3.1: ByTAM: Services and architecture

3.3 Monitoring System

The Monitoring System consists of an infrastructure of sensors that continuously collect data from the environment (eg. network devices, resources, ...), from the service currently operating (latency, load, message size, etc.), and delivers this data to the Adaptation Manager for storage and processing. Tolerance to faulty sensors is obtained by having multiple replicas of each sensor and by performing a voting on its outputs. Therefore, at least $3f + 1$ sensor replicas are needed to tolerate f failures. In this way, each sensor is seen as a SMR client and the received updates obtained by the sensors are treated as commands which are totally ordered by the BFT protocol.

As a result, the sensors produce a linear sequence of captured values identified by tuples which, in addition to the collected data, include a unique identifier of the replica/sensor and a sequence number. These tuples are stored in the Adaptation Manager. However, individual values are not used directly. Instead, ByTAM waits for a quorum of $\lceil \frac{(n+f)}{2} \rceil$ different sensor readings. Since the sensor readings are totally ordered, every correct replica will receive the same ordered value sequence from the replicated sensors. Therefore, a deterministic function can be applied to delete f highest and lowest (extremes) values in order to extract a middle value. A limitation of this approach is that a consensus execution is needed for each value produced by the sensor. A simple approach to attenuate this cost consists in grouping different readings and execute the consensus on that group.

3.4 Adaptation Manager

The Adaptation Manager is responsible for processing the data obtained from the Monitoring System, register and evaluate the adaptation policies, and start the adaptation process. The Adaptation Manager has three services, as illustrated in Figure 3.1(b): storage, policy management, and adaptation coordinator. Essentially, the Adaptation Manager processes the data that comes from the sensors and computes quality metrics. These metrics are then evaluated by policies that were previously installed. These policies analyze the metrics obtained and decide if a reconfiguration action is needed. Like the

Monitoring System, the Adaptation Manager is a Byzantine fault tolerant replicated service. It is important to remark that the messages exchanged between the Monitoring System and the Adaptation Manager are digitally signed, therefore non-authenticated sensor messages are discarded.

3.4.1 The storage service

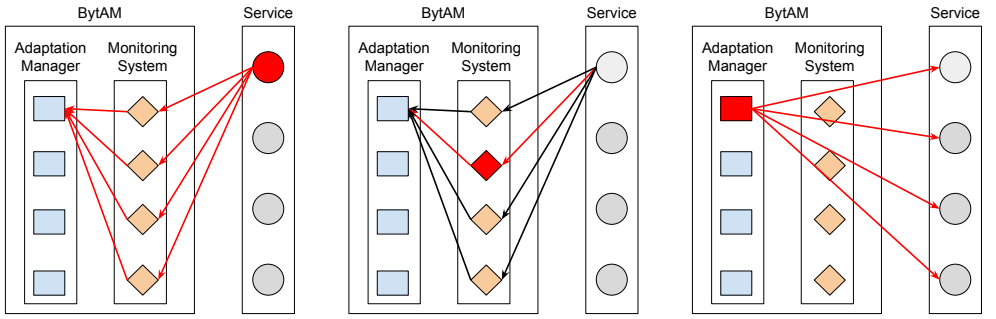
The storage services stores the data collected by each sensor. As explained in Section 3.3, the Monitoring System produces a linear sequence of values; therefore, each correct replica of the Adaptation Manager eventually obtains the same sequence of values in sufficient number to be processed or a prefix of this sequence (which will inevitably receive the remaining values following the correction properties of *consensus*). Therefore it is possible to ensure consistent decision making across all correct replicas, because the policies will evaluate exactly the same system state. By allowing the continuous collection of the information retrieved by the sensors, we allow a policy to be written with access to the history of retrieved values. This may be useful because isolated readings may be subject to transient fluctuations for which the cost of starting a reconfiguration does not compensate. Additionally, it becomes possible to detect operational patterns over time (and adapt in a pro-active manner). Despite the ability to develop these kind of policies with ByTAM, the development of specific policies that consider the cost-benefit and tendencies has not been done in the context of this dissertation and is delegated to future work.

3.4.2 The policy manager

The policy manager controls the life cycle of the adaptation policies. More specifically, it allows listing, installation/removal of policies, associate/disassociate and activate/deactivate policies of a specific Managed System. The definition of the adaptation policies will be described in detail in Section 3.7. The currently active policy to perform the adaptation is registered in a configuration file using a unique identifier associated with each policy. When it is time to execute an adaptation, the policy with the unique identifier present in the configuration file is loaded and executed, allowing the dynamic switching among different pre-loaded policies. In the future we aim to extend the system in order to allow the dynamic loading of new/updated policies. This flow must be performed by an utility which changes the configuration of the Adaptation Manager in a coherent manner. Dynamic policy loading provides greater flexibility when compared with other solutions since it allows loading new policies even while the system is operating.

3.4.3 The Adaptation coordinator

The Adaptation coordinator is responsible for the configuration of a given Managed System – it establishes a group of initial operational parameters, adds/removes replicas from a Managed System, defines active replicas and backups, etc. Furthermore, the adaptation coordinator executes the active policies while supporting different evaluation criteria (periodic, reactive, personalized) and, if a reconfiguration is necessary, sends commands in order to reconfigure the Managed System.



(a) Managed System Failure (b) Monitoring System Failure (c) Adaptation Manager Failure

Figure 3.2: Messages from failed replicas on different components ($f = 1$)

3.5 Performing an Adaptation

On the previous paragraphs, we showed how the Adaptation Manager worked and how the replicas decided unanimously on coherent representations of the system. Still, it is important to remark that replicas need to agree on when a given policy must be executed while agreeing, respectively, on the version of the policy that must be applied (guaranteeing timely and consistent adaptations). As mentioned, a policy can be executed while reacting to an event, such as a surpassing of a limit defined by the policy from the obtained values sent by the sensors. In any case, we assume that that all policy activations are executed consecutively. We also assume that a serial number is associated to each policy activation. Each policy activation increments this identifier and disseminates the command ADAPT($i+1, s$) to every other replica, where s is an identifier of the state on which the policy activation $i+1$ must be performed. Furthermore, a replica which receives $f+1$ ADAPT commands from different replicas, also adopts the decision of activating the policy upon the state s, disseminating also the command ADAPT($i+1, s$) to itself. An Adaptation Manager replica executes the policy when it receives $2f+1$ equal ADAPT commands. The messages exchanged between Adaptation Manager replicas are also digitally signed. So, commands from non-authenticated replicas are not considered for the quorum validation. Ultimately, if the policy sends reconfiguration commands to the Managed System, it will only execute the reconfiguration if it receives this command from a quorum of Adaptation Manager replicas.

3.6 Dealing with Byzantine Components

As it was previously explained, each ByTAM component operates independently, communicating exclusively through message exchange. This eases both the analysis of the system behavior when faults occur and the mechanisms that avoid the propagation of these failures, as it is illustrated in Figure 3.2. In the following paragraphs we discuss the effects of the occurrence of faults on different components (namely the Managed System, the Monitoring System and the Adaptation Manager).

When one of the replicas of the Managed System exhibits Byzantine behavior, the Monitoring Sys-

tem readings can obtain arbitrary values or fail silently. In the example provided in Figure 3.2(a), the Managed System replica can emit intentionally different values for each sensor in order to force the activation of a specific policy and consequently reduce the system availability during the reconfiguration interval. To avoid the propagation of arbitrary values produced by a Byzantine replica of the Managed System, each Monitoring System replica must read the same indicator across all Managed System replicas and perform a fault tolerant voting on the values read, before propagating the reading result to the Adaptation Manager. However, most replicas of the Managed System will send coherent values to the Monitoring System replicas for which all the correct replicas will end up sending a correct reading to the Adaptation Manager.

When a Monitoring System component fails, as it is illustrated in Figure 3.2(b), it may either not provide the readings to the Adaptation Manager or, change arbitrarily the values read and send contradictory values to different Adaptation Manager replicas. The propagation of contradictory values is filtered by the consensus execution (when the Adaptation Manager receives the values). A reading omission is treated in the same way as an incorrect value. Finally, incorrect values which are received by the Adaptation Manager are eliminated in the voting process (which uses the values received by other Monitoring System replicas): the value of the reading is either greater/smaller than the values produced by other replicas or it is within the interval defined by the values received by two correct replicas. On the first scenario, the deterministic function deletes the extreme values in the voting process. On the second scenario, if the value is within the interval defined by two correct replicas, then it is also a correct value. This strategy is similar to what is used in other contexts, such as in (Dolev, Lynch, Pinter, Stark, and Weihl 1986).

Finally, when an Adaptation Manager replica fails (Figure 3.2(c)), it can either opt to not execute the policy or trigger an incorrect adaptation at an inappropriate time. However, these behaviors will be masked by the remaining Adaptation Manager replicas. It is important to recall that every Adaptation Manager replica receives the same sequence of values sent by the Monitoring System, as described earlier. In this way, each correct Adaptation Manager replica individually applies the policies and, if needed, trigger an adaptation. Since the policies are deterministic, every correct replica of the Adaptation Manager will trigger the same adaptation. Hence, even if one of the Adaptation Manager replicas cuts communication with the Managed System, the remaining Adaptation Manager replicas will end up sending the reconfiguration command, triggering the adaptation process. Even if a Byzantine replica sends an incorrect command to the Managed System, it will not start the adaptation immediately. Instead, the Managed System waits for a quorum of $f + 1$ equal reconfiguration commands from different replicas to perform the adaptation.

The analysis previously made assumes that different replicas of a given component fail independently. This forces a careful selection from the machines where the replicas are executed. However, there is no inconvenience in co-locating replicas from different components on the same machine (eg. a Monitoring System replica with an Adaptation Manager replica) as long as no more than f replicas fail for each service. Finally, the managing components, such as the Adaptation Manager or the Monitoring System, can be shared to perform a dynamic adaptation of different Managed Systems, since the adaptation frequency is typically low.

3.7 Adaptation Policies

The policies accepted by the Adaptation Manager are specified in the *event-condition-action* (ECA) form. To create a policy, it is necessary to implement the Adaptation Manager API and specify which events activate the policy, which conditions must be verified upon its activation and, finally, which action must be executed to apply an adaptation.

The events that activate a policy may be notification or temporal events. So, a policy can be registered to be executed when a specific value from the Monitoring System is received, usually for a quick response to a definitive event, for instance, a crash of a Managed System replica which executes the PBFT (allowing the reconfiguration of the replication factor for a smaller number of replicas).

In a similar way, we can use heuristics to predict the behavior of some dynamic variables while knowing the history of the metrics previously registered. It is then possible to elaborate a periodically activated policy which selects a range of readings obtained from the Managed System. For instance, it is possible to use the storage to update the training set of a system based in automatic learning such as ADAPT.

The development of new policies goes through the implementation of a *Policy* interface which is provided by the Adaptation Manager. This interface provides two methods: *trigger()* and *execute()*.

On the *trigger()* method, we describe the events that stimulate the execution of the policy, that is, this method represents a previous phase of the policy's execution where we compute if the execution is necessary. If the policy's execution is necessary, then a signed ADAPT command is sent to all the other Adaptation Manager replicas. After receiving $2f + 1$ valid ADAPT commands from different replicas, the *execute()* policy method is called.

On the *execute()* method, a new configuration is created which is then sent to the Managed System. This configuration uses the reconfiguration API provided by the currently active Managed System. When the Managed System receives $f + 1$ valid signed messages from different Adaptation Manager replicas, it executes the reconfiguration described in that message.

Summary

In this chapter we have been through the design and implementation of ByTAM. We explained the three different components of ByTAM and how they interact with each other. We also demonstrated situations where a failure occurs on each component of ByTAM. Finally we have explained what it is needed for implementing new policies in order to adapt a specific managed BFT system.

In the next chapter we present the experimental evaluation made using this prototype.

4 Evaluation

ByTAM supports the dynamic adaptation of any reconfigurable parameter exported by the managed system. In this section, we intend to illustrate the advantages of having an independent and flexible system to control the adaptation, instead of using solutions that only support a restrict number of adaptations which are hardcoded. Particularly, we use a scenario where we increase the performance not only by changing the degree of fault tolerance (resilience), but also one the parameters of the currently active BFT protocol. None of the previous systems offers the flexibility to perform this kind of adaptation.

4.1 Experimental Setup

The replicas and clients used in the evaluation of ByTAM were hosted on the DigitalOcean service (https://www.digitalocean.com/). Each replica/client has its own virtualization environment (achieved through the KVM). Each virtualized environment has access to 512Mb of RAM, gigabit ethernet connections between switches and 10 Gigabit (ethernet) connections to internet providers, Intel Xeon E5-2630L v2 processors clocked at 2.40Ghz (6 cores) and 20Gb SSDs. Each replica also has a unique IP address associated. The automatic adaptation system is based on BFT-SMaRt therefore, its components are executed on a Java Virtual Machine (JVM). The JVM version used was 1.8.0_91. Each replica started with an initial heap size of 256Mb and used the G1GC as the algorithm for garbage collection.

To evaluate the performance of each configuration, we used a payload on the BFT system, generated by client machines with various threads. To achieve that we used benchmarking tools offered by the BFT-SMaRt library, more precisely the *ThroughputLatencyServer* and the *ThroughputLatencyClient*. Each request to the replicated service and each received response has a size of 512 bytes. Each client thread sends 5000 sequential requests to the replicated service without any delay between them. During the calibration phase for the evaluation, we verified that a single machine executing clients did not generate sufficient requests to overload the service but, no more than two machines are required to strangle the service. So, every experiment was performed using two client machines, while changing the number of client threads executed on each one, between 1 to 15 execution threads in each one (to a total of 30 client threads).

4.2 The evaluated Adaptation

The adaptation used in the evaluation corresponds to the reconfiguration *scaleDown* of the policy presented on Listing 4.1. This policy describes two possible outcomes of the system when a replica

```
public class AdaptQuorumSize implements Policy {
  private Server server = null;

  @Override
  public void trigger(Events evts) {
    if (evts.has(Event.ReplicaDown)) {
      Server newServer = getBackupServer();
      if (newServer != null) {
        this.server = newServer;
      }
      sendAdaptMsg(); //send notification to GA to run the policy
    }
  }

  @Override
  public void execute() {
    if (server != null) {
      integrate(server);
    } else {
      scaleDown();
    }
  }

  private void scaleDown() {
    Configuration newConfig = new Configuration();
    newConfig.set("f","1");
    newConfig.set("n","4");
    newConfig.set("checkpoint","100");
    sendConfigMsg(newConfig); //send reconfiguration commands to SG
  }
    ...
}
```

Listing 4.1: Example of an Adaptation Policy for ByTAM.

fails. If there are more freely available machines in the system, a new machine is integrated with the currently active replicas. If there are no machines available to replace a failed replica then, it is preferable to reconfigure the system to only tolerate one additional failure (changing the configuration to "$f = 1$" and "$n = 4$"). However, while we perform the reconfiguration to the group of replicas, it is also possible to reconfigure the configuration parameters of the protocol in order to optimize its performance on the new configuration. Particularly, one of the parameters used by BFT-SMaRt is the checkpoint frequency of the state history. The experiences that we have performed show that, for the used load, and for a configuration with "$f = 2$" and "$n = 7$" the ideal value for the checkpoint is 1000 stored operations while for a configuration with "$f = 1$" and "$n = 4$" the ideal value is 100 stored operations (these values were obtained experimentally by running the system with different values for this parameter with intervals of 50 units). ByTAM does not only allow the reconfiguration of the protocol but also the number of active replicas.

4.3 Results

We assess the performance of the system when, on an initial configuration with "$f = 2$", "$n = 7$" and checkpoint with 1000 stored operations, a replica fails ("$f = 1$", "$n = 6$"). In this case, we evaluated three possible alternatives: to leave the system working with that configuration until a new replica becomes available to replace the failed one (note however that the configuration still tolerates an additional failure); reducing the number of servers of the service to "$f = 1$" and "$n = 4$", maintaining the checkpoint frequency used by the protocol (this configuration still supports an additional failure); reducing the number of servers and adjust the checkpoint configuration, as it is captured in Listing 4.1.

To measure the performance on the various configurations, we have monitored the throughput and

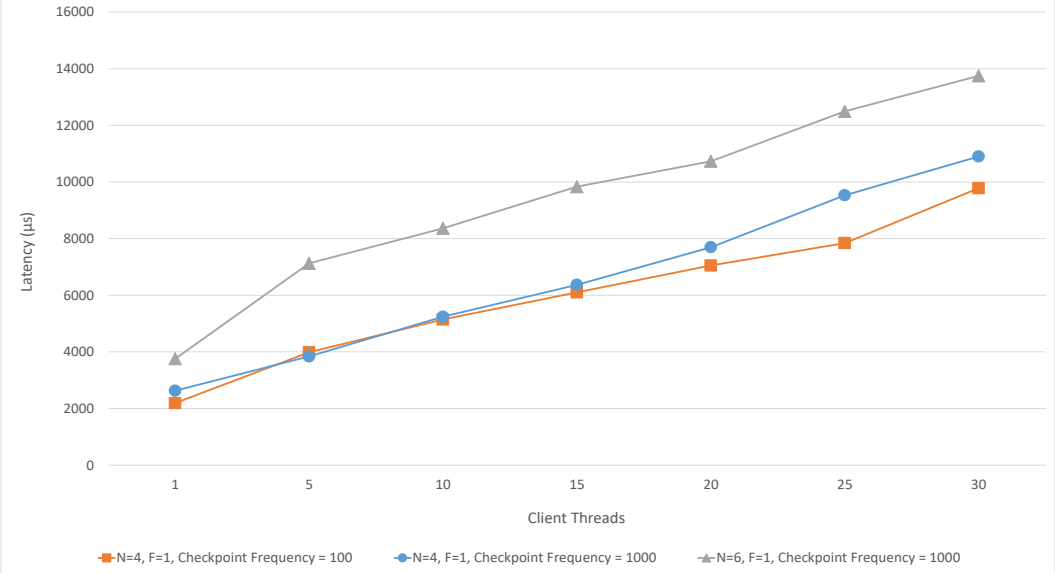


Figure 4.1: Latency of the various configurations

the latency of the system on the three configurations already described. The throughput captures the number of executed operations per second while the latency captures the time interval from sending a client request and the reception of the respective response (in μs).

The results are shown in Figures 4.1 and 4.2, with an increasing number of clients performing requests on the replicated service. As it is shown in Figure 4.1, the configuration which has the lowest latency is the configuration where the policy described in Listing 4.1 was applied. Additionally in Figure 4.2, it is shown that the same configuration also exhibits the best overall throughput. We conclude that between the three configurations provided, the configuration with "$f = 1$", "$n = 4$" and checkpoint frequency of 100 operations, offers the best service to its clients (high throughput and low latency) therefore, performing the adaptation improves the service provided. A big advantage of ByTAM is that the decision of performing these adaptations together (changing both the number of replicas and checkpoint frequency) or separately, according to other criteria, can be easily changed through a policy re-write, without changing any code from the adaptation manager or from the managed system, which did not happen on previous systems.

We have also registered how long it took for the system to perform some reconfiguration tasks. We used the policy described in Listing 4.1 so the values retrieved are from reconfiguration tasks which are relevant for that policy. More specifically, we measured how long it took to i) create a new configuration for the target system. This includes retrieving relevant sensor values registered by the Adaptation Manager, create a new configuration using the Managed System reconfiguration API, and digitally signing the message; ii) reconfigure the checkpoint frequency of the Managed System (with 4 and 7 active replicas); iii) reconfigure the resilience of the system – reducing the number of active replicas from "$n = 7$" to "$n = 4$"; iv) reconfigure both the resilience and the checkpoint frequency of the Managed System.

As for the number of consensus instances performed by the Adaptation Manager, since each value from the the Monitoring System is totally ordered, we have one consensus execution for each sensor value. So, the number of consensus executions is between $n - f$ and n (where n is the number of

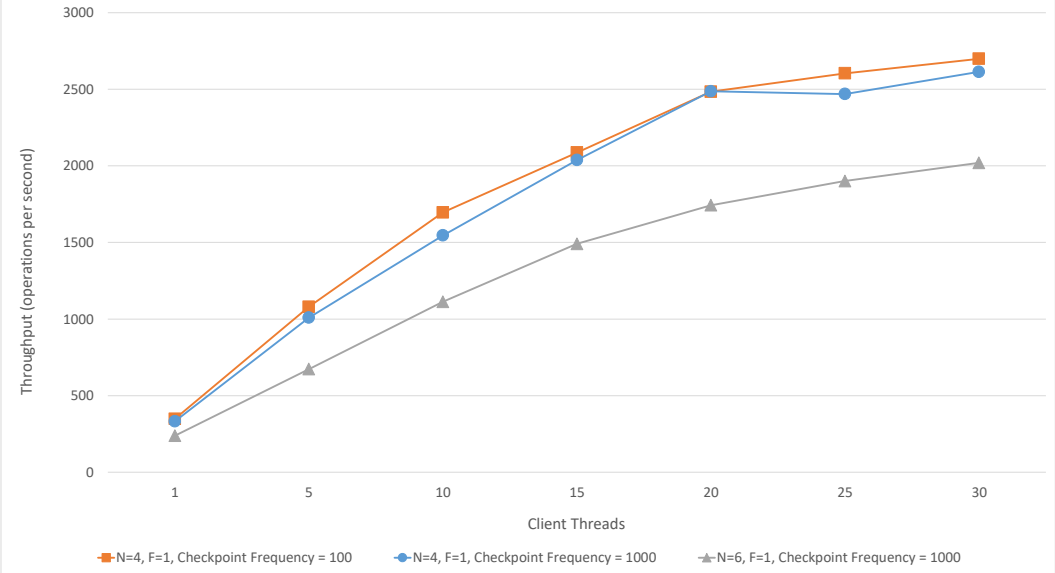


Figure 4.2: Throughput of the various configurations

Task	Average Time (in ms)
Creation of the message containing the new configuration by the policy	13
Changing checkpoint frequency (with $n = 4$)	29
Changing checkpoint frequency (with $n = 7$)	54
Changing resilience ($n = 7$ to $n = 4$)	165
Changing checkpoint frequency and resilience ($n = 7$ to $n = 4$)	246

Table 4.1: Execution time for reconfiguration tasks described in Listing 4.1

replicas of the Monitoring System). The Adaptation Manager also sends an ADAPT message to all the other Adaptation Manager replicas when a given replica wants to run the currently active policy (as it is described in Section 3.4). Since these messages are also totally ordered, a consensus instance is executed for every correct Adaptation Manager replica which is between $n - f$ and n.

Summary

In this chapter we introduced the experimental evaluation made to ByTAM and its results. We also demonstrated a scenario where a replica is faulty and an example policy to respond to that event. We have measured the performance of the solution (with and without reconfiguration) with a variable number of clients. Finally we have measured how long it took to perform the different steps of the reconfiguration.

The next chapter finishes this thesis by presenting the conclusions regarding the work developed and also introduces some directions in terms of future work.

5 Conclusions and Future Work

In this dissertation we present ByTAM, a robust architecture to perform dynamic adaptations of systems that tolerate Byzantine faults. To our knowledge, ByTAM is the first reconfiguration system which is open source and tolerates Byzantine faults across all system components, either from replicas from the managed system, from the adaptation manager, or from sensors that capture the state of the system. Additionally and unlike previous proposals, ByTAM supports the execution of multiple adaptation policies without requiring changes to the managed system or to the adaptation manager. The current ByTAM version can be obtained on `https://github.com/fmrsabino/library/tree/bytam`.

ByTAM is flexible enough to execute a wide range of adaptation actions, as long as these are supported by the system being adapted. An early goal of this work was to experiment with adaptations that would change the algorithm used by the target BFT-system. Unfortunately, I have not been able to implement the required changes to BFT-SMaRt. As a future work we would like to augment BFT-SMaRt to support the concept of abortability such that it becomes possible to perform the dynamic reconfiguration of the BFT algorithm. Additionally, we would also like to extend ByTAM to support the injection of new policies into the system without having to restart it (as of now, only the dynamic exchange of policies already installed is allowed).

References

Abd-El-Malek, M., G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie (2005). Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review 39*(5), 59–74.

Aiyer, A. S., L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth (2005). Bar fault tolerance for cooperative services. In *ACM SIGOPS Operating Systems Review*, Volume 39, pp. 45–58. ACM.

Bahsoun, J.-P., R. Guerraoui, and A. Shoker (2015). Making BFT protocols really adaptive. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 904–913.

Bessani, A., E. Alchieri, M. Correia, and J. Fraga (2008). Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 163–176.

Bessani, A., J. Sousa, and E. Alchieri (2014). 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)*, pp. 355–362.

Bessani, A. N., M. Santos, J. Felix, N. F. Neves, and M. Correia (2013). On the efficiency of durable state machine replication. In *USENIX Annual Technical Conference*, pp. 169–180.

Cachin, C., R. Guerraoui, and L. Rodrigues (2011). *Introduction to reliable and secure distributed programming*. Springer.

Castro, M. and B. Liskov (2000). Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pp. 19–19. USENIX Association.

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

Clement, A., E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti (2009). Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, Volume 9, pp. 153–168.

Cowling, J., D. Myers, B. Liskov, R. Rodrigues, and L. Shrira (2006). Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 177–190. USENIX Association.

Dolev, D., N. Lynch, S. Pinter, E. Stark, and W. Weihl (1986, May). Reaching approximate agreement in the presence of faults. *J. ACM 33*(3), 499–516.

Dwork, C., N. Lynch, and L. Stockmeyer (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*(2), 288–323.

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

Guerraoui, R., N. Knežević, V. Quéma, and M. Vukolić (2010). The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pp. 363–376. ACM.

Kotla, R., L. Alvisi, M. Dahlin, A. Clement, and E. Wong (2007). Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review 41*(6), 45–58.

Lamport, L. (1978a). The implementation of reliable distributed multiprocess systems. *Computer Networks (1976) 2*(2), 95–114.

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

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

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

Pease, M., R. Shostak, and L. Lamport (1980). Reaching agreement in the presence of faults. *Journal of the ACM (JACM) 27*(2), 228–234.

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

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)*, San Francisco, California, pp. 189–204.