Policy-Based Adaptation of Byzantine Fault Tolerant Systems

Miguel Neves Pasadinhas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor:
Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson:	Prof. Luís Manuel Antunes Veiga
Supervisor:	Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee:	Prof. João Carlos Antunes Leitão

October 2017

Acknowledgements

I would like to thank my advisor Professor Luís Rodrigues and Professor Antónia Lopes, for their guidance throughout this work. I would also like to thank my friends for all the leisure moments we shared during this year. Last, but definitely not least, I would like to thank my parents for the endless support of each of my decisions.

Lisbon, October 2017

Miguel Neves Pasadinhas

For my parents and my cats,

Resumo

Ataques maliciosos, falhas de *hardware* ou até mesmo erros de operadores podem fazer com que um sistema se comporte de forma arbitrária e difícil de prever. Tolerância a faltas Bizantinas (BFT) engloba um conjunto de técnicas para tornar um sistema resiliente à presença de faltas arbitrárias. Vários protocolos BFT foram propostos na literatura, cada um otimizado para diferentes condições operacionais. Este facto levou ao desenvolvimento de alguns sistemas BFT adaptativos, capazes de se ajustar às condições atuais. Infelizmente esses sistemas não possuem mecanismos expressivos para especificar políticas de adaptação. Para além disso, sistemas com mecanismos mais expressivos para especificar tais políticas não possuem algumas abstrações fundamentais para a adaptação de sistemas BFT.

Neste contexto, apresentamos nesta dissertação uma linguagem para especificação de políticas de adaptação de sistemas tolerantes a faltas Bizantinas. Para além disso, apresentamos um motor robusto que, dado um ficheiro com a política de adaptação escrita na linguagem proposta, é capaz de decidir quais as melhores adaptações por forma a guiar um sistema gerido num caminho em conformidade com os seus objetivos.

Abstract

Malicious attacks, hardware failures or even operator mistakes may cause a system to behave in an arbitrary, and hard to predict manner. *Byzantine fault tolerance* (BFT) encompasses a number of techniques to make a system robust in face of arbitrary faults. Several BFT algorithms have been proposed in the literature, each optimized for different operational conditions. For that reason, adaptive systems able to adapt BFT systems to the current operational conditions have been proposed but unfortunately all lack expressive mechanisms to specify adaptation policies. Other systems provide expressive mechanisms to specify those policies but lack important abstractions for BFT systems' adaptation.

Considering this context, in this thesis we present an adaptation policy specification language that targets Byzantine fault tolerant systems. In addition, we present a robust engine that, given a policy written in the proposed language, is able to decide the best adaptations to guide a managed system in a path of accordance with its business goals.

Palavras Chave
Keywords

Palavras Chave

Tolerância a Faltas Bizantinas

Sistemas Adaptativos

Adaptação Dinâmica

Políticas de Adaptação

Especificação de Linguagens

Algoritmos de Decisão

Keywords

Byzantine Fault Tolerance

Adaptive Systems

Dynamic Adaptation

Adaptation Policies

Language Specification

Decision Algorithms

Contents

List of Figures

List of Algorithms

Acronyms

BFT Byzantine Fault Tolerant

SMR State Machine Replication

KPI Key Performance Indicator

MAC Message Authentication Code

1 Introduction

Byzantine fault tolerance encapsulates a set of techniques used to build robust systems, able to cope with a minority of replicas failing arbitrarily. These techniques usually rely on some Byzantine fault tolerant consensus protocol.

This thesis addresses the problem of adapting Byzantine fault tolerant systems. In particular, it presents Policaby: a robust adaptation manager able to decide which adaptations better suit the current executional envelope of a managed system, guiding it in a path of accordance with its business goals. In order to decide which are the best adaptations, Policaby uses a novel, tailor made language for adaptation policy specification able to capture the available adaptations and the business goals of the managed system.

1.1 Motivation

Due to the growth of cloud and on-line services, building dependable, fault tolerant systems is becoming more challenging. This happens because cloud and on-line services are more complex and more difficult to protect. Thus, they are more prone to malicious attacks, software bugs or even operator mistakes, that may cause the system to behave in an arbitrary, and hard to predict manner. Faults that may cause arbitrary behaviour have been named Byzantine faults, after a famous paper by Lamport, Shostak, and Pease(Lamport, Shostak, and Pease 1982). Due to their nature, Byzantine faults may cause long system outages or make the system behave in ways that incur in significant financial loss.

State-machine replication(Lamport 1978) is a general approach to build distributed fault-tolerant systems. Implementations of this abstraction that can tolerate Byzantine faults are said to offer *Byzantine fault tolerance* (BFT) and typically reply on some form of Byzantine fault-tolerant consensus protocol.

The exact number of replicas that are needed to tolerate f Byzantine faults depends on many system properties, such as on whether the system is synchronous or asynchronous, what type of cryptographic primitives are available, etc. In most practical settings, $3f + 1$ replicas are needed to tolerate f faults. Thus, BFT protocols are inherently expensive. It is therefore no surprise that after the algorithms proposed in (Lamport, Shostak, and Pease 1982), a significant effort has been made to derive solutions that are more efficient and can operate on a wider range of operational conditions, such as(Castro and Liskov 2002; Abd-El-Malek, Ganger, Goodson, Reiter, and Wylie 2005; Cowling, Myers, Liskov, Rodrigues, and Shrira 2006; Kotla, Alvisi, Dahlin, Clement, and Wong 2010; Clement, Wong, Alvisi, Dahlin,

and Marchetti 2009).

Unfortunately, despite the large number of solutions that exist today, there is not a single protocol that outperforms all the others for all operational conditions. In fact, each different solution is optimised for a particular scenario, and may perform poorly if the operational conditions change. For instance, Zyzzyva(Kotla, Alvisi, Dahlin, Clement, and Wong 2010) is a BFT protocol that offer very good performance if the network is stable but that engages in a lengthy recovery procedure if a node that plays a special role (the leader) is suspected to have failed.

Given this fact, researchers have also started to design adaptive BFT systems, that can switch among different protocols according to the observed operational conditions. One of the first systems of this kind is Aliph(Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015). An important contribution of Aliph was that it introduced a precise property that BFT protocols must exhibit in order to support dynamic adaptation, named *abortability*. On the other hand, the number of possible adaptations, and the sequencing of those adaptations, is very limited in Aliph: the system is only able to switch among a small set of pre-defined protocols in a sequence that is hardcoded in the implementation. Other authors have extended the approach in an attempt to support a richer set of adaptations, and to provide the user more control on how the system adapts, depending on the specific business goals of a given deployment(Bahsoun, Guerraoui, and Shoker 2015). Still, as will be discussed later in this thesis, even those approaches suffer from several limitations.

An adaptation policy is a specification that captures the goals that the system aims at achieving and that guides the selection of the adaptations that are better suited to achieve those goals under different conditions. The goal of this thesis is to identify the right abstractions to express those adaptation policies in the context of Byzantine fault tolerant systems and develop a robust adaptation manager that implements those abstractions.

1.2 Contributions

This thesis describes a new language to specify adaptation policies alongside an implementation of a robust engine able to parse that language and make informed decisions on which set of adaptations better suit the current executional envelope. In addition, it provides an experimental evaluation of the engine's performance. As a result, this thesis makes the following contributions:

- Proposes a novel adaptation policy specification language, able to capture the business goals of a managed system and which operations (adaptations) are available for choosing.
- Details an engine able to parse the proposed language and therefore make an informed decision of which adaptations better suit the current conditions.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A specification (including a complete grammar) for a novel adaptation policy specification language.
- Policaby, an implementation of the proposed engine.
- An experimental evaluation of Policaby's decision algorithm performance.

Research History

This work benefited from the fruitful comments of Professor Antónia Lopes, Carlos Carvalho and Bernardo Palma.

A paper that describes parts of this work has been accepted for publication in Actas do Nono Simpósio de Informática, INForum 17 (Pasadinhas, Porto, Lopes, and Rodrigues 2017).

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

1.4 Structure of the Document

The remaining of this document is organized as follows. Chapter 2 provides an introduction Byzantine fault tolerance and the state of adaptive systems. Chapter 3 describes the assumed system architecture, the novel adaptation policy specification language's specification and the details of Policaby's engine. Chapter 4 presents the results of the experimental evaluation of Policaby's engine. Lastly, Chapter 5 concludes this document by summarizing its main points and discussing a few directions for future work.

2 Related Work

This chapter gives an overview of previous work addressing adaptation Byzantine Fault Tolerant systems. It starts by describing some of the existing BFT protocols, to motivate the need for changing the protocol according to the conditions of execution environment. Next it surveys previous adaptive BFT systems and overviews BFT-SMaRt, one of the few open source libraries supporting BFT state machine replication, and ByTAM, an early prototype that attempted policy-based adaptation of Byzantine Fault Tolerant systems. For completeness, at the end of the chapter, a number of systems that, although not targeting BFT systems specifically, use interesting techniques for adaptation are surveyed.

2.1 Byzantine Fault Tolerance

This subsection gives an overview of four different BFT protocols, namely: the original protocol by Lamport, Shostak, and Pease(Lamport, Shostak, and Pease 1982), PBFT(Castro and Liskov 2002), Zyzzyva(Kotla, Alvisi, Dahlin, Clement, and Wong 2010) and Aardvark(Clement, Wong, Alvisi, Dahlin, and Marchetti 2009). It starts by the original formulation provided in (Lamport, Shostak, and Pease 1982) even if these protocols only work in synchronous systems, then addresses PBFT as it is one of the first BFT algorithms to exhibit acceptable performance on asynchronous systems; Zyzzyva and Aardwark are variants of PBFT that optimize its operation for different operational conditions.

2.1.1 Byzantine Generals Problem

The term Byzantine Fault Tolerance was introduced by Lamport, Shostak, and Pease(Lamport, Shostak, and Pease 1982). In that paper the authors identified a key problem to achieve BFT, namely the need to reach consensus in the presence or arbitrary faults, that they have named the *Byzantine Generals Problem*. The problem has been illustrated as follows:

Several divisions of the Byzantine army, each commanded by its own general, are surrounding an enemy city. After observing the city, they must agree on a common plan of action. The generals may only communicate by messenger. The problem becomes harder as some of the generals are traitors and try to prevent the loyal generals from reaching an agreement, for example, by sending different messages to different generals. A correct solution to this problem must guarantee that:

1. All loyal generals decide upon the same plan of action.

2. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

The authors then proved that this problem can be reduced to an equivalent one in which a commander general must send an order to his $n-1$ lieutenants such that (1) all loyal lieutenants obey the same order and (2) if the commanding general is loyal then every loyal lieutenant obeys the order he sent. From now on, we will refer to the commander general as the *primary* and the lieutenants as *backups*. Also, we will refer to either the primary or backups as nodes.

In that paper, two protocols to solve Byzantine consensus are presented. Both protocols assume a synchronous system but make different assumptions about the cryptographic primitives used, namely in what regards the ability to sign messages. The protocols are elegant but rely on a recursive design that makes them exhibit a high message overhead. For simplicity in the algorithms specification, we will refer to "obtaining a value" instead of "obeying an order".

The first protocol assumes that nodes can communicate through messages that are delivered correctly and the absence of a message can be detected. Furthermore, a receiver of a message knows who sent it. As nodes can detect the absence of a message, we will ignore the case in which a node does not send a message, as a previously defined default value can be used by the nodes who should have received that message. The algorithm is a recursive function $OM(m)$ that solves the problem of a primary sending an order to its $n-1$ backups if, at most, m out of $n = 3m+1$ total nodes are faulty. Since the algorithm is a recursive function, we start by defining the stopping condition as $m = 0$. In that case, the primary sends a value to every backup and every backup uses that value. In the recursive case, (1) the primary sends his value to all backups, then (2) for each backup i, let v_i be the value that it received from the primary; backup i acts as the primary in $OM(m-1)$, sending v_i to the other $n-2$ backups; finally (3) backup i uses $majority(v_1, ..., v_n)$, where $v_j, j \neq i$ is the value received from backup j in step (2). $majority$ is a deterministic function that necessarily chooses a majority value if one exists, for instance, the median of the set of values. Note that as m increases, the number of messages sent by this protocol will also increase due to the recursive nature.

The second protocol makes the same assumptions about messages with two extra ones: (1) signatures from correct nodes cannot be forged, changes made to the content of their signed messages can be detected and (2) anyone can verify the authenticity of signatures. This removes complexity to the problem, as a node can no longer lie about values received by other nodes. Before the algorithm begins, each node i has an empty set V_i. Initially, the primary signs and sends its value to all backups. Each backup, upon receiving a signed message m from the primary, if no message as been received yet, sets V as the set containing the received value v and then signs m and sends it to all other backups. If a backup i receives a message m from another backup containing a value $v \notin V$, it adds v to V; if m is not signed by all backups, then backup i signs it and sends it to all backups that did not sign m. Eventually, backup i will no longer receive messages as all values are signed by all nodes. At that point, it uses a function $choice(V)$ that deterministically chooses a value from V (or a default value if $V = \emptyset$). All correct nodes use the same $choice(V)$ (for instance the median value) and all correct nodes end up

with the same set V, therefore choose the same value.

Both protocols are very simple but rely on assumptions that are not usually true in distributed systems. In particular, these algorithms require *synchronous systems* where we can set an upper bound to messages delay and clocks are synchronized. These assumptions do not hold on an *asynchronous* distributed system like the *Internet*. Next, we will discuss BFT algorithms that can operate on asynchronous systems.

2.1.2 PBFT

The Practical Byzantine Fault Tolerant protocol(Castro and Liskov 2002) (PBFT), proposed by Castro and Liskov, was one of the first BFT algorithms to exhibit an acceptable performance and to operate on asynchronous systems. This algorithm tolerates at most $f = \lfloor \frac{n-1}{3} \rfloor$ faults from a total n replicas. It assumes an asynchronous network which may omit, duplicate, delay, or re-order messages. Nodes may fail arbitrarily but independently from one another. In order to guarantee that the nodes fail independently, the authors suggest that each node should run different implementations of the service and of the operating system.

The protocol works as follows. The client sends a request to all replicas. Replicas exchange several rounds of messages among them to ensure coordination, even in the presence of corrupted replicas. When agreement is reached among replicas, a reply is sent back to the client. Coordination among replicas is achieved using a distributed algorithm where one of the replicas plays a special role, known as the *primary* (if the primary is faulty, the protocols is able to make progress by forcing the change of the primary). A failure free run of the coordination protocol, depicted in Figure 2.1, includes the following steps:

1. The primary replica multicasts a PRE-PREPARE message
2. When a replica receives a PRE-PREPARE message it multicasts a PREPARE message
3. When a replica receives $2f + 1$ PREPARE and/or PRE-PREPARE messages it multicasts a COMMIT message
4. When a replica receives $2f + 1$ COMMIT messages it executes the client request and sends a REPLY message to the client

When the client receives $f + 1$ REPLY messages with the same result, from different replicas, accepts the result. A sketch for the proof that the thresholds above guarantee safety can be found in (Castro and Liskov 2002) and a detailed proof is given in (Castro and Liskov 1999).

If the primary replica fails, an algorithm to select a new primary must be executed. That will allow the system to make progress, guaranteeing liveness. According to the terminology used in the PBFT paper, a view defines a set of replicas where one is designated the primary and the others are designed

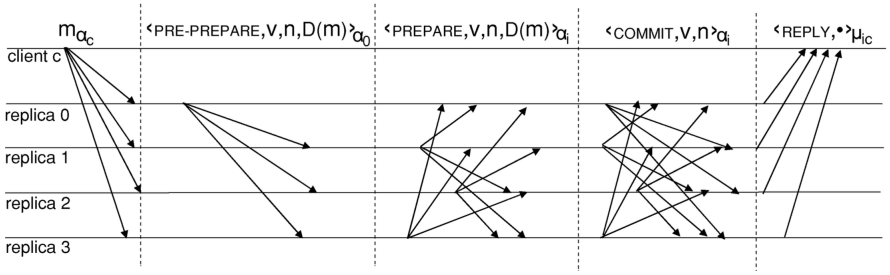

Figure 2.1: PBFT failure free run. Image taken from (Castro and Liskov 2002)

as backups. Therefore, selecting a new primary involves running a procedure that is called a *view change*. The procedure is triggered as follows. Let v be the current view. When a backup is waiting for a request in view v (i.e. it received a valid request but has not executed it) it starts a timer. If the timer expires the backup stops accepting messages (except view change related messages) and will multicast a VIEW-CHANGE message for view $v + 1$, containing information about its state. When the primary replica of view $v + 1$ receives $2f$ VIEW-CHANGE messages for view $v + 1$ from other replicas, it sends a NEW-VIEW message to all replicas, making sure every replica starts the new view at the same time and with the same state.

2.1.3 Zyzzyva

Zyzzyva(Kotla, Alvisi, Dahlin, Clement, and Wong 2010) is a BFT protocol that uses speculation as a means to reduce the cost of BFT. In this protocol, the client sends a request to the primary which assigns a sequence number and forwards the ordered request to the replicas. Upon receiving an ordered request, the replicas speculatively execute the request and respond to the client. The idea of the protocol is that consistency only matters to the client, therefore it is the client's responsibility to verify if the responses are consistent. The responses consist of an application response and an history. This history allows the client to detect inconsistencies in the request ordering.

When waiting for responses, the client sets a timer and the protocol's path will be dictated by the number and consistency of replies it receives before the timer runs out. Two replies are considered consistent if the application response is the same and the history is compatible. We will overview the possible scenarios in the next paragraphs.

The first case, whose message pattern is illustrated in Figure 2.2, is known as the *fast case*. If the client receives $3f + 1$ consistent replies then the request is considered complete and the response is delivered to the application. At this point there is a guarantee that, even in the presence of a view change, all correct replicas will execute the request in the same order, producing the same result. Sketches of the proofs can be found in the paper(Kotla, Alvisi, Dahlin, Clement, and Wong 2010).

If some replicas are faulty or slow, it can happen that the client does not receive $3f + 1$ consistent replies. However, if the client still manages to receive at least $2f + 1$, it is still possible to make progress without forcing a view change, by execution one more step; this is known as the *two-phase case*. In this

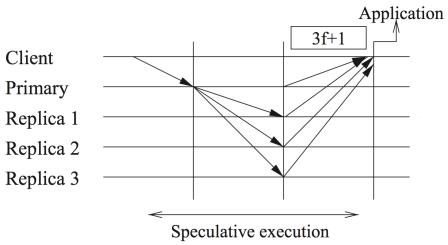

Figure 2.2: Message pattern of Zyzzyva's fast case. Image taken from (Kotla, Alvisi, Dahlin, Clement, and Wong 2010)

case, depicted in Figure 2.3, the client has a proof that a majority of the replicas executed the request consistently. The replicas, however, may not have that proof. Therefore, the client assembles a *commit certificate* that sends to all replicas. The commit certificate is a cryptographic proof that a majority of correct servers agree on the ordering of all the requests up to this request inclusive. Upon receiving the commit certificate, the servers send an acknowledge message. If the client receives $2f + 1$ (a majority) of acknowledgements, it delivers the response to the application. Otherwise the case described in the next paragraph is triggered.

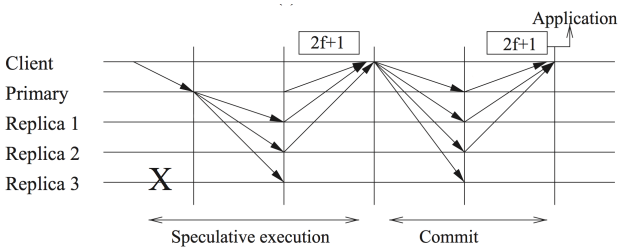

Figure 2.3: Message pattern of Zyzzyva's two-phase case. Image taken from (Kotla, Alvisi, Dahlin, Clement, and Wong 2010)

The last case is known as the *view change case*. This case is more complex. If the client receives less than $2f + 1$ consistent responses, it retransmits the request, this time to all replicas, which in turn forward the request to the primary for sequencing. After sending the ordering request to the primary, replicas set a timer. If the timer expires without a response from the primary, replicas start a view change. If a replica receives a sequence number for the request, it will speculatively execute it and send the response to the client. If the client detects valid responses in the same view with different sequence numbers it will send them as a *Proof of Misbehaviour* to all replicas. Upon receiving the Proof of Misbehaviour, replicas start the view change protocol.

The speculative execution of Zyzzyva performs better than PBFT when the primary is not faulty. However, when the primary is faulty, some of the work performed optimistically is lost and has to be re-executed in a conservative manner, yielding a longer worst-case execution.

2.1.4 Aardvark

The BFT protocols discussed above follow a different flow of execution in the non-faulty and faulty-scenarios. The optimizations made for the fault-free scenarios often introduce extra complexity when faults occur or even open the door for the existence of runs where a single Byzantine fault may cause the system to become unavailable. The authors of Aardvark(Clement, Wong, Alvisi, Dahlin, and Marchetti 2009) advocate that a BFT protocol should provide an acceptable performance even in the presence of at most f Byzantine replicas and an unbounded number of Byzantine clients. With the aim of achieving the desired performance even under faults, the design of Aardvark follows a number of principles that, according to the authors, go against the "conventional wisdom".

The first principle is the usage of signed messages. Signing a message is a computationally expensive process, therefore not used in most BFT protocols. Instead, the protocols tend to use a Message Authentication Code (MAC) to authenticate messages. This technique is based on shared symmetric key therefore less expensive than signed messages that require asymmetric keys. As a consequence of the usage of symmetric keys, MACs do not provide non-repudiation of messages. The absence of non-repudiation causes extra complexity on BFT systems. Aardvark's approach is to require clients to sign their requests. This allows the replicas to trust that a request was indeed issued by some client, making the BFT protocol more robust.

The second principle is to use view changes as a normal operation, and not only as a last resort. Aardvark performs view changes on a regular basis. The performance of the primary replica is monitored by other replicas, and compared with a threshold of minimal acceptable throughput (that is adjusted dynamically). If the primary fails to meet that throughput requirement, the other replicas start a view change. This limits the throughput degradation of the system in the presence of a faulty primary, as it will need to be fast enough, otherwise it will be replaced by a new primary.

The last principle is related to resource isolation. Aardvark uses a set of network interface controllers (NIC) which connect each pair of replicas. As a consequence, each communication channel becomes independent from each other, and faulty replicas cannot interfere with the timely delivery of messages from correct servers. This also allows replicas to disable specific NIC, protecting themselves from attacks. Other protocols do not adopt this technique as it would decrease their performance due to the loss of hardware-supported multicast. Aardvark also uses distinct queues for client and replica messages, so that client requests cannot prevent replicas from making progress.

The message pattern of Aardvark is very similar to PBFT, as it also uses a three-phase commit protocol. The main differences are the three aspects discussed above, which make Aardvark the protocol (of the ones discussed in this Section) with worst performance in a fault-free environment but more robust in the presence of faults.

2.2 Adaptable Byzantine Fault Tolerant Systems

As discussed, each of the three practical protocols surveyed above is tuned for different system conditions. To the best of our knowledge, there is no single BFT protocol able to outperform all others under each system conditions. It is therefore fundamental to be able to adapt the system.

When reasoning about system adaptations there are three fundamental questions that should be answered: *what to adapt*, *when to adapt*, and *how to adapt*. Each of these questions will be discussed individually.

What to adapt refers to the set of possible adaptations that the system can perform. In the adaptation of BFT systems there are two main classes of possible adaptations – changes to the replica set and changes to the replicas' configuration. When adapting the replica set it is possible to, at least, change the total number of replicas or replace replicas for newer ones (for instance, to replace faulty or old replicas). When adapting the configuration of a given replica, there are two main categories of adaptations:

- Changing the configuration of an internal module: this allows to tweak internal parameters of some module of the system, for instance, change the quorum size of the consensus algorithm(Sabino 2016).
- Replace a module: allowing for changing the implementations of a given module. One example is switching the BFT protocol that the system is using(Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015; Bahsoun, Guerraoui, and Shoker 2015).

When to adapt is the problem of choosing the instant in which the system should perform an adaptation. There are two main approaches:

- *Periodically*. In this case the system sets a timer. When the timer expires the system verifies its policies and tries to find an adaptation (or set of adaptations) that can improve the system, i.e. makes the system closer to some business goals.
- *Based on predefined thresholds*. This approach triggers the adaptation process when some threshold or condition is violated or met.

Despite being conceptually different approaches, both are usually triggered by some kind of event – either the expiration of the timer or a change in the system conditions. Most of the systems analysed in this Chapter tend to adapt based on predefined thresholds and conditions. Such conditions may be as simple as $latency > 10ms$ or more complex conditions for instance: a server failed(Sabino 2016), there is contention(Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015) or even some protocol will have at least a 10% better performance than the current one(Bahsoun, Guerraoui, and Shoker 2015). The techniques proposed in (Rosa, Rodrigues, Lopes, Hiltunen, and Schlichting 2013) also contemplate the approach of periodically trying to improve some chosen business goals.

The problem of *how to adapt* consists of selecting the techniques that allow the system to change some configuration with minimal performance loss during the transition. Changes to the replica set will most likely involve some sort of view change protocol. Changing the BFT protocol can be done multiple ways, for instance, (1) stopping one and starting the other(Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015) or (2) having both running at the same time during a transition period until the older is definitely removed(Mocito and Rodrigues 2006). It is possible that some adaptations can be done without coordination, for instance, changing the batch size of the Consensus protocol may only require a change in the primary replica. This problem is, however, considered to be out of the scope of this thesis.

2.3 The Next 700 BFT Protocols

The observation that different protocols perform better in different conditions has motivated the authors of (Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015) to study techniques that allow to replace a protocol in runtime in a safe manner. In this work, the authors propose: i) a novel abstraction to model a SMR protocol that can abort its execution to support reconfiguration; ii) several instances of protocols that support such abstraction, and; iii) some simple policies to dynamically commute among these protocols. We briefly discuss each of these contributions in the next paragraphs.

2.3.1 Abstract

In their work, the authors of (Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015) have identified a set of properties and interfaces that allow one protocol to be aborted during its execution and to retrieve from the aborted protocol enough information to allow operation to resume using a different protocol. A protocol that owns this set of features is denoted to be an *instance* of a generic protocol simply called *Abstract*. An adaptive BFT protocol may support several of these instances, although only one should be active at a given point in time. When a reconfiguration takes place, the active instance is aborted and a new instance is activated. State is transferred from the old instance to the new instance through a request history that is handed to the client when the instance is aborted.

2.3.2 Instances

The authors consider different protocol instances that are compliant with Abstract, namely Backup, ZLight, Quorum, and Chain.

Backup is an Abstract instance that guarantees exactly k requests to be committed. k is a configurable parameter and must be greater or equal to 1. Backup is described as a wrapper that can use any other full-fledged BFT protocol as a black-box. In this paper, the authors used PBFT as the Backup

protocol due to its robustness and extensive usage. This means that Backup will ensure that exactly k requests will be committed and then abort to the next instance.

ZLight is an instance that mimics Zyzzyva's fast case. This means that ZLight makes progress when there are no server or link failures and no client is Byzantine. When those progress conditions are not met, meaning the client will not receive $3f + 1$ consistent replies, the client will send a PANIC message to the replicas, triggering the abortion of this instance.

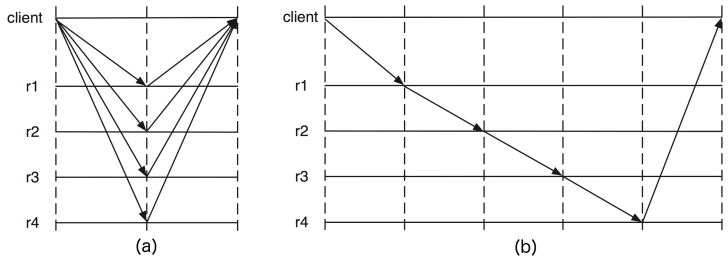

Figure 2.4: Message pattern of (a) Quorum and (b) Chain. Image taken from (Aublin, Guerraoui, Knežević, Quéma, and Vukolić 2015)

Quorum is a protocol with a very simple message pattern, depicted in Figure 2.4(a). It has the same progress conditions as ZLight (there are no server nor link failure and no Byzantine clients) plus an additional one: there must be no contention. This one round-trip message pattern gives Quorum a low latency but at the cost of not relying on total order of the requests (therefore the need for the extra progress condition). In this protocol, the client sends the request to all replicas that speculatively execute it and then send the reply to the client. If the client does not receive $3f + 1$ consistent messages it starts the panicking mechanism (just like in ZLight) causing the current instance to abort.

Chain is a new protocol based on the one proposed in (van Renesse and Schneider 2004) (which only tolerated crash), adapted to tolerate Byzantine faults. As can be seen in Figure 2.4(b), Chain organizes the replicas in a pipeline and every replica knows the fixed ordering of the replicas. The first replica in the pipeline is the *head* and the last one is the *tail*. Replicas only accept requests from their predecessor (except the head which accepts requests from the client) and only send messages to their successor (except the tail which sends the reply to the client). Due to their novel authentication method, *chain authentication*, they are able to tolerate Byzantine faults. This instance has the same progress conditions as ZLight (no server nor link failures and no Byzantine client). If the client does not receive a reply or verifies the reply is not correct (by using the chain authentication mechanisms) it starts panicking, triggering the abortion of the instance.

2.3.3 Policies

The authors have experimented to build adaptive BFT protocols based on the instances above. Two variants of the system have been developed, namely AZyzzyva (that only uses ZLight and Backup) and Aliph (that uses Quorum, Chain, and Backup). In both cases, they use very simple policies to switch among protocols. The available instances are totally ordered in a directed ring and an adaptation always

replaces one instance by its successor in the ring. Each instance is characterized by a set of conditions required for progress; when these conditions are not met, an adaptation is triggered.

2.3.4 Discussion

Abstract allows the creation of protocols that make progress under very specific conditions. This allows for very simple protocols able to have very good performance under those conditions. For instance, Quorum allows Aliph to have a very low latency, due to its one round-trip message pattern, if there are no faults and no contention. Under contention but still with no faults, Chain has a good throughput, allowing Aliph to handle periods where multiple clients send many requests to the system. As a fall-back, Backup will ensure progress under faulty environments. Despite those advantages, Aliph lacks a mechanism to express the adaptation policies. The policies are hardcoded in the Abstract instances in the form of the progress conditions, making it very hard to configure. Also, usage of a static ordering for the protocols is a downside of their approach.

2.4 Adapt

Adapt is an abortable adaptive BFT system. It monitors the system conditions in order to decide – with the help of mathematical formulas and machine learning – which protocol would have the best performance. It is composed by three sub-systems: BFT System (BFTS), Event System (ES) and Quality Control System (QCS). BFTS is the actual system to be adapted and includes the abortable BFT protocols implementations. The Event System is responsible for monitoring the BFTS, collecting *Impact Factors* – chosen metrics which have a significant impact on the system. The authors chose the number of clients, request size, response size, and faulty replicas as the impact factors. The ES sends periodic information to the QCS with changes in the impact factors. The QCS is the decision maker in Adapt, and therefore the sub-system in which we are more interested in.

The Quality Control System (QCS) algorithm works as follows. Let's assume that we have a set of n BFT protocols to chose from (implemented in BFTS) and some protocol $p_i, i \in [1, n]$ is currently running. The QCS is triggered by the reception of an event from the ES, stating some change in the impact factors. Such event starts the *evaluation process*, that selects a protocol $p_j, j \in [1, n]$, which is the best protocol for current system conditions. If $j \neq i$ and p_j offers a significant improvement over p_i then the QCS orders the BFTS to switch the protocol to p_j.

Most of the logic in the QCS is in the *evaluation process*. Before we can study the evaluation process in detail, we must define two types of metrics, namely *Key Characteristic Indicators* (KCIs) and *Key Performance Indicators* (KPIs). KCIs encode the static characteristics of a protocol (e.g whether it tolerates faulty clients or requires IP multicast); KCI values are defined before the system starts and are static during its execution. KPIs are dynamically computed metrics that evaluate the performance

of a protocol (e.g. latency and throughput); KPI values are computed several times in runtime, using prediction functions, as described below.

The evaluation process consists of a static part and a dynamic part, using the KCIs and KPIs respectively. The static part consists in excluding from the adaptation process all protocols that are considered not suitable by the user. For this purpose the user is required to input her *user preferences*, in the form a vector that states which KCIs a protocol must have in order to be eligible as a valid adaptation target. All protocols that do not match the user preferences vector will be excluded from the final result. Remember that the KCIs for each protocol are manually defined before the system starts. In the dynamic part, Adapt uses the KPIs values to predict which of the protocols will have the better performance among them. Because KPIs can have different natures, they are divided into type β^+ and type β^-. β^+ KPIs are said to have an high tendency, meaning an higher value means a better evaluation, for instance throughput. On the other hand, β^- KPIs are said to have a low tendency therefore lower values mean a better evaluation. Latency is an example of a β^- KPI. Despite that, KPIs can have very different values, possibly by many orders of magnitude. To solve that problem the KPI values for each KPI are normalized to the interval $[0, 1]$. This normalization will result in a value of 1 to the best value of a given KPI, 0 to the worst one and the remaining values are linearly interpolated. Which are the best/worst values depend on whether the KPI is of type B^- or B^+. For B^- KPIs the best value will be the minimum value and the worst will be the maximum value. For the B^+ KPIs will be the opposite.

At this point in the evaluation process, each protocol has a value for each KPI in the range $[0, 1]$. As a result of normalizing, the values lost some meaning and can now only be used as a ranking between the protocols. A value of 1 does not mean that the KPI is at its maximum value, it just means that the protocol which has that value is the best among the evaluated ones. The final evaluation of a protocol corresponds to the sum of each associated KPI value multiplied by a weight. A weights vector – assigning a weight to each KPI – must be specified by the user before the system starts. The selected protocol will be the one, not excluded by the KCIs, having the biggest evaluation score.

Adapt supports a form of *heuristics*. In Adapt, an heuristic is a vector just like the weight vector. If an heuristic is present, each component of the weight vector is multiplied by the corresponding component of the heuristic vector resulting in a new weight vector. This could be used, for example, to give more weight to throughput and less weight to latency under high contention.

The KPI values are computed in runtime using prediction mechanisms, in this case Support Vector Machines for Regression(Smola and Schölkopf 2004). The described process of predicting a single KPI is a very generic Machine Learning process. First, the protocol runs for a period of time and the KPIs under each state are recorded. After a sufficient number of records are gathered, called a *training set*, a *prediction function* is trained with it. The prediction function takes impact factors as inputs and outputs a KPI value. The objective of the training phase is to tune the prediction function parameters so that it gives the most accurate predictions. Once the prediction function is trained, when the ES sends new values for the impact factors, it is executed to predict the values used for KPIs in the evaluation process.

The authors of Adapt also mention that in order to have the most accurate predictions the training set must be updated by the ES so that the prediction function improves itself periodically.

As mentioned above, Adapt only switches protocol if the new protocol has a significant improvement over the previous one. Switching protocol comes with a cost so if the gain is small maybe switching is not the best solution. The authors define S_{thr}, a minimum improvement threshold required for the protocol to be switched. So the protocol switching is bounded to the following restriction: $\frac{p_{max}}{p_{curr}} \geq S_{thr}$, where p_{max} is the evaluation value of the best (selected) protocol and p_{curr} the evaluation value of the current protocol.

This analysis of Adapt is concluded with a discussion of its strong and weak points. Adapt was a good step forward in the adaptation of BFT systems. It introduced several improvements over Aliph, for example:

- The new protocol is chosen by analysing system conditions instead of a predefined hardcoded order
- It can change protocol based on KPIs
- It uses Machine Learning to improve the prediction of the best protocol under each system conditions

Despite those improvements, Adapt still lacks a good way to specify adaptation policies. Those policies can be specified using the *user preferences* vector, which excludes some protocols from being selectable, the *weights vector* and *heuristics*.

The *user preferences* vector does not focus the business goals of the system and only an expert in BFT can configure it. A better approach could be to have some way of letting the system decide that some protocol x could not be chosen because it violated some business rule.

When populating the *weights vector*, it is not obvious what weights to give to each KPI. This problem is aggravated by the fact that the values are normalized, therefore lost some meaning. As mentioned above, a protocol with a value 1 for some KPI only means that it is the better value among the evaluated ones – it can still be a bad value overall. It is not clear what giving a 0.7 weight to throughput and 0.3 to latency would mean, as the normalized values do not represent the actual value. It is therefore really hard to come up with a good weights vector.

The *heuristics* are not described in great detail. It is not clear how their trigger can be specified, therefore we cannot discuss that part. Despite that, they have the same problems as the weights vector, as they have similar impact. The need for this kind of heuristics in Adapt might be product of poor policy specification methods used, as the authors felt the need of changing the system objective under different conditions.

2.5 BFT-SMaRt

BFT-SMaRt(Bessani, Sousa, and Alchieri 2014) is an open-source library for creating BFT SMR systems. It implements a protocol similar to PBFT and is, according to the authors, the first of its kind to support reconfigurations of the replica set. During its development, simplicity and modularity were big concerns. Optimizations that could create unnecessary complexity in the code by introducing new corner cases were avoided. Unlike PBFT which implements a monolithic protocol with the consensus algorithm embedded in the SMR protocol, BFT-SMaRt has a clear separation of both modules. As a consequence, this system is easier to implement and reason about. There are also modules for state transfer and reconfiguration.

The API is very simple and due to the modular approach it is also extensible. In order to implement deterministic services the server must implement an `execute(command)` method and the client simply invokes it with an `invoke(command)` call. All of the BFT complexity is encapsulated between those method calls.

2.6 ByTAM

ByTAM(Sabino 2016) is an early prototype of a BFT Adaptation Manager built using BFT-SMaRt. ByTAM assumes an architecture similar to Adapt, having a *monitoring system*, a *managed system* and an *adaptation manager* (AM). Our biggest interest is in the AM, which is responsible for the policies.

The AM stores the data collected from the monitoring system. The policies can then consume that data. Storing all of the sensors data could allow for complex policies that use the historical data to predict the system's behaviour or whether the current conditions are just transient fluctuations. Despite that potential, the implemented policies in ByTAM do not explore that path.

The AM is also responsible for managing the policies and checking when an adaptation should be triggered. Policies are defined in the Event-Condition-Action form. The event can either be periodic or some message from the monitoring system. When an event triggers a policy, its condition is checked. If it passes, the system executes the action. Typically, the action consists in sending a message to the managed system, telling it what to adapt, but it could also be tuning an internal prediction engine model, for instance.

The policies are Java classes which implement a specific interface. As all Java code is accepted in the policy, the potential of this form of policies is tremendous. However the low level expressiveness also makes the policies much harder to write and more error prone. The policy specifier will have to handle Java's complexity instead of being focused on the business goals of the system. Despite that potential for creating very complex policies, only a very simple policy was implemented (upon a failure, it integrated a new replica in the system or adjusted the quorum size, if there was no spare replica). This could be a result of the difficulty in expressing policies.

2.7 Some Relevant Techniques used in Adaptive Systems

In this Section we will survey some techniques used adaptive systems that do not target BFT specifically. Although these systems and techniques were not designed with BFT in mind, their ideas may be useful when designing a new BFT Adaptation Manager. We will overview a new language able to describe impact models which can cope with non-determinism. Next we will study a work related with learning and refining impact functions. Finally, we conclude this Section by discussing a technique which looks at policies as a set of ordered goals the system should fulfil.

2.7.1 Impact Models Under Uncertainty

Cámara et al. defend in (Cámara, Lopes, Garlan, and Schmerl 2016) that fully deterministic models of the system behaviour are not attainable in many cases, so it is crucial to have models able to handle the uncertainty. They propose a language to define impact models which can handle uncertainty. Furthermore, they explain how those impact models can be used to formulate adaptation strategies and how to predict the expected utility of those strategies.

The approach relies on the availability of an architectural model of the system, defined as a set of components and connectors (with associated properties). For instance, components can be nodes (servers and clients) and connectors the links among them (for instance a TCP connector). Those components and connectors can have associated managed and monitored properties. For instance, a server can have a managed property that captures which BFT protocol it is currently executing and a monitored property *load* representing the server load.

Adaptation actions are the adaptive primitives implemented in the system. For each adaptation action, an impact model should be defined. The language proposed in this paper uses Discrete Time Markov Chains in order to define impact models able to handle different outcomes. Each impact model is defined as a set of expressions in the form $\phi \rightarrow \alpha$, where ϕ is a guard condition and α is a probabilistic expression. At any time, at most one guard condition can be true for each impact model.

The probabilistic expressions are the ones responsible for handling the uncertainty. They can define different outcomes with different probabilities. For instance, if some expression α is of form $\{[0.7]\{\alpha_1\} + [0.3]\{\alpha_2\}\}$, then α_1 is estimated to occur 70% of the times and α_2 only 30%. α expressions can be simpler, for example, `forall s:Servers | s.load' = s.load / 2` would mean the load of each server would drop to half of the current one. Note that the symbol `'` is used to denoted the new value of the property.

An adaptation strategy is a composition of several adaptation actions. Each strategy has a applicability condition and a body. The applicability condition dictates whether the strategy can be applied and the body is a tree. Each edge has a guard condition, an adaptation action and a success condition. The success condition is used to evaluate whether the adaptation action succeeded and its value can

be used in edges that leave the achieved node. This allows the definition of complex policies in the form of adaptation strategies, which are specified in Stitch(Cheng and Garlan 2012).

When a system needs to adapt, there may be several applicable strategies. In order to chose which one to execute, the system must have a goal to achieve. The goal must be defined in such a way that it can be related to runtime conditions, such as response time and cost. To each of those we must specify:

- An utility function, that assigns a value in the range $[0, 1]$ to the runtime condition in question, being 1 the better value.
- A utility preference which is a weight given to the runtime condition in question. The sum of all the weights must be 1.

In order to assess the utility of a strategy, the system creates a probability tree of each possible outcome. When compared to the strategy tree, this probability tree has more branches because of the non-determinism – anywhere an adaptation strategy would be used we have a branch for each possible outcome. The utility of the strategy is said to be the sum of the utility of each possible state multiplied by its probability.

The techniques presented in this paper provide a solution for handling non-determinism when specifying impact functions of adaptations. This helps coping with the uncertainty of the outcome of an adaptation. Despite that, it is still difficult to discover and give a probability to all impact functions. In Section 2.7.2 we will study machine learning techniques that help mitigate this problem.

2.7.2 Learning Adaptation Models Under Non-Determinism

Francisco Duarte proposed in his master's thesis a system able to revise impact functions using machine learning techniques – Ramun(Duarte 2016). It supports impact models that include non-determinism, like the ones presented above. Ramun works as follows:

1. Initial impact models are collected from system experts
2. A synthetic dataset is created with fabricated samples that reflect the initial impact models
3. New samples are added to the dataset by monitoring the effect of adaptations, with the system running
4. The extended dataset is analysed by the K-Plane(Bradley and Mangasarian 2000) algorithm
5. The result is processed and written to a text file

The resulting text file is compliant with the language proposed in (Cámara, Lopes, Garlan, and Schmerl 2016). The K-Plane algorithm returns k planes, each mapping to an impact function. The larger the k value is, the more impact functions the algorithm will output and more accurate the model

will be. In order to achieve a balance between the number of impact functions and the model's accuracy, Ramun iteratively runs the algorithm with an increasing k. The initial k is 1 and the final k will be the one that provides a model with an error inferior to a configurable threshold T.

After the final k planes are computed, their valid input ranges are defined. Each plane – and therefore each impact function – is only valid in the range where it has samples. Non-determinism comes into play whenever multiple planes share the same input region for every variable. Geometrically, this means that the hypercubes formed by their input regions intersects. Although this is hard to imagine for n-dimensions, it is easy in two dimensions. Imagine two impact function that take x as input and return x', the new value for x. If for the same x we have two possible x' values (one given by each impact function), then we have non-determinism. The probability of each case is calculated as the number of samples that that case has in the region, over the total number of samples in the region.

The techniques presented in Duarte's thesis help minimizing one of the problems in choosing the right adaptation – predicting its impact. This work provides a solution for learning even in the presence of non-determinism. Another great feature is that its output can be translated into a language able to be read by humans, making it easier for a system administrator to understand what the algorithms are learning.

2.7.3 Rank-based Policies and Adaptations

Rosa et al. (Rosa, Rodrigues, Lopes, Hiltunen, and Schlichting 2013) proposed a language to specify policies as set of *goals*, that are ordered in decreasing degree of relevance. The system strives to satisfy as many goals as possible, starting from the most relevant, i.e., the system sees if there is a set of adaptations that can satisfy the most important goal; if yes, it then, among these adaptations, selects those that can also satisfy the next goal, and so forth.

The goals are specified in terms of *key performance indicators* (KPIs) such as *cpu usage* or *latency*, for instance. Two types of goals can be defined relative to the KPIs, namely *exact goals* and *optimization goals*. Exact goals specify *acceptable* ranges for the KPIs. The ranges are specified with either of three primitives – *Above*, *Below* and *Between*, meaning the KPI value should be above some threshold, below some threshold and between two thresholds, respectively. Optimization goals specify *desirable* values for the KPIs. Those goals can state that a KPI should be maximized, minimized, or approach some target value. Each optimization goal has an assessment *period*, and every period the system will try to optimize that goal.

The system is able to select the set of adaptations that match a given policy. For that purpose, adaptations have a number of attributes such as, among others, a guard condition (stating under which conditions it can be executed) and impact functions (stating the expected changes in the KPIs). In (Rosa, Rodrigues, and Lopes 2011), node adaptations were proposed as a mechanism to deal with distributed systems. However, the set of possible actions over distributed system's nodes was very limited. To give

an example in the context of this thesis, it was not possible to specify an adaptation that changed the leader replica of a BFT protocol.

The system has an *offline* phase and an *online* phase. During the offline phase, adaptation rules are generated. This rules are inferred from the goals and adaptations already defined. For instance, given a goal stating that *cpu_usage* should be below 0.45 (and assuming we are interested in 0.01 granularity), the associated rule is triggered when an *event* stating the *cpu_usage* KPI is at least 0.46 is received. The goal of a rule is always to increase or decrease the value of a KPI. Adaptations that drive the KPI in a direction contrary to the intended one are excluded from the set of possible adaptations of that rule.

During the online phase, when a rule is triggered, the system evaluates which of the valid adaptations better suits the situation. It is in this phase that the ranking of the goals is relevant. Each goal has a rank associated with the order in which it is defined. The first goal is the most important and the last one is the least important. Each goal will work as a filter of adaptations. Given a set S of possible adaptations, each adaptation is evaluated against the most important goal. Adaptations that violate that goal are excluded from S. The chosen adaptation will be the last to violate a goal. If all adaptations violate some goal k, all remaining goals are still used for tie-breaking.

This approach provides an elegant way of specifying objective goals that the system should fulfil. The usage of a ranking based goal specification allows for a graceful degradation of the system, allowing us to ensure maximum effort to preserve certain properties. This can be useful when adapting a BFT system as we may want to guarantee, for instance, that we can tolerate Byzantine faults as a primary goal. Also, this approach does not force the system manager to give unnatural weights to KPIs.

Summary

This chapter introduced Byzantine fault tolerance and some well known BFT algorithms. As discussed, each algorithm is fine tuned to have a better performance in some specific case and to the best of our knowledge, no single algorithm is able to outperform all other under all conditions. For that reason, BFT adaptive systems were introduced.

Some of the most relevant BFT adaptive systems – such as Aliph and Adapt – were studied. Despite their effort and positive results, the adaptation policy expression mechanisms used by those systems are weak and do not allow much configuration. On the other and, some of the languages and systems for adaptation policy specification that do not target BFT systems specifically lack some important abstractions useful for this kind of systems.

3 Policaby

This chapter presents Policaby, a reliable adaptation manager with a novel language for adaptation policy specification. Section 3.1 presents the system architecture assumed by Policaby, Section 3.2 details the novel adaptation policy specification language made for Policaby and finally Section 3.3 describes important protocols and algorithms used by Policaby during execution.

3.1 System Architecture

Policaby is an adaptation manager designed to decide adaptations that guide a BFT managed system in a path of accordance with its business goals. Policaby is itself designed to be able to tolerate Byzantine faults. In order to decide which adaptations better suit the managed system at each moment, it is necessary to have metrics that characterize the execution environment. For that, Policaby relies on metrics gathered by a monitoring system. The monitoring system is responsible for managing a set of sensors able to collect data about the managed system and aggregating the sensor data into useful metrics. The metrics are then used by the Policaby's engine to make an informed decision about which adaptations better suit the current executional envelope.

Policaby makes the following assumptions regarding the monitoring system:

- It provides a Byzantine fault tolerant service that exposes metrics' values.
- It provides a Byzantine fault tolerant timer service that, upon a request from Policaby, notifies it when the requested timer expires.

Since we are building a reliable, Byzantine fault tolerant, adaptation manager, all correct Policaby's replicas must decide the same adaptations. For that to be possible, all correct replicas must execute using the same data. However the monitoring system is constantly gathering data. We need to ensure that all Policaby's replicas have access to the exact same values for each metric. To achieve that, the metrics' values service should be versioned and, if all Policaby's replicas use the same version *id* to access it, then all will use the same data, making the same decision (given that Policaby's decision algorithm is deterministic, which it is). The problem of all replicas using the same data was reduced to the problem of all replicas using the same version *id* to query metrics. This is solvable using the timer service.

Policaby's execution is triggered by the reception of a timer event by the monitoring system's timer service. That timer message must have the *id* to be used to query the metrics.

As an optimization, Policaby's replicas were designed to execute on the same process as the monitoring system's replicas. This prevents consensus executions and reduces the latency of metrics requests. Each Policaby's replica communicates only with the monitoring system's replica hosted by the same process. If monitoring system's replica is not correct, than Policaby's replica won't be correct as it will trust the data provided by it. On the other hand, if the monitoring system's replica is correct, it will provide correct data to Policaby and all correct Policaby's replicas will produce the same results. This means that Policaby's replicas do not need to coordinate with each other, avoiding the usage of consensus protocols (however the monitoring system still needs to use consensus before sending messages to Policaby's replica, to agree on a version *id*, for instance). Since faulty replicas can produce incorrect results, the managed system must ensure that it only executes an adaptation if it received that request from a big enough quorum of Policaby's replicas.

3.2 Policaby's Language

One of the main goals of Policaby was the design of an expressive language to specify adaptation policies. This section details the novel language designed for Policaby, firstly by giving an overview example of the language and then by detailing each construct available. When detailing the language's constructs, a EBNF grammar will be provided for each. We recommend using the grammar as a tool to understand how exactly the described constructs can be used in the language. For reference, non-terminal symbols are surrounded by angle brackets, strings that must be matched are surrounded by single quotes, terminal symbols start by lower case. A special symbol – \n – is used to denote a new line. For reading convenience, most required white spaces have been ignored (it may be required, for instance, between strings and terminal symbols). The modifier *?* means zero or one occurrence, *+* quantifies at least 1 occurrence and * is matched by 0 or more occurrences.

3.2.1 Overview

Most BFT protocols resort to a leader replica and typically the leader's performance has a significant impact on the overall system's performance. Therefore a relevant adaptation would be to choose a better leader for the protocol. For instance, if a BFT system is using Zyzzyva, in the fast case, the replicas' communication always involves the leader (i.e. there are no backup replicas communicating with one another). Selecting the replica with the least latency to the remaining replicas to be leader would therefore have a positive impact on the time taken to coordinate the processes. Let's name that time *t_coord* and assume it is inversely proportional to the leader's latency to the other replicas (a simplification made for ease of exposition). Let's also assume that the leader changing algorithm has a 10% chance of failing, resulting in no changes in the system.

A complete specification of the case above in Policaby's language could be:

```
System:
    Params:
        leader: Replica
        protocol: Protocol
        t_coord: number

Enum Protocol Instances Zyzzyva, PBFT

Component Replica:
    Params:
        mensurable latency: number

Adaptation changeLeader:
    Input:
        r: Replica
    Requires:
        leader != r
        protocol = Zyzzyva
    Impacts:
        [0.90]:
            t_coord *= r.latency / ledaer.latency
            leader = r
        [0.10]:
            leader = leader

Goal Minimize t_coord
```

Listing 3.1: An example of a policy file.

In this specification, concepts such components, adaptations and goals can be spotted. This section gives a brief explanation of each but in depth details will be given in dedicated sections. The specification starts with a *System* construct. That construct allows the specification of global system parameters. For instance, in the example, the system has a parameter *leader* of type *Replica*, another named *protocol* of type *Protocol* and a number *t_coord*.

The types *Replica* and *Protocol* are defined immediately below. In this language, types are defined as *Components* which can have *Instances* and parameters associated with each instance. If we draw a parallel with object oriented programming languages, components would be classes. The *Enum* construct is just syntactic sugar for defining a Component with no parameters and the specified instances. Instances of components can be created and destructed as the result of executing an adaptation.

An *Adaptation* encapsulates an operation and its expected effects in the system. Adaptations can

be parameterized with components using the *Input* construct. During runtime, each combination of inputs will be associated with that adaptation. We call this process *adaptation instantiation*. This means that a single adaptation specification can be instantiated several times, resulting in multiple adaptation instances. In this example, the *changeLeader* adaptation receives a Replica as parameter. When Policaby instantiates this adaptation, there will be an adaptation instance for each Replica instance in the system. Each adaptation instance would therefore represent changing the leader to a specific replica. The adaptations must specify its *Impacts*. This are predicted changes in some key metrics of the system. This impacts can have associated probabilities, allowing us to deal with the non-determinism associated with adaptations' impacts.

Lastly a single goal is defined for the system: minimizing the coordination time. As we can see the specified adaptation is said to have impact in *t_coord*, therefore the adaptation which gives the lowest value for *t_coord* would be selected. Note that in this case there is a single and simple goal, making it easy to identify which adaptation instance would be selected. However, in the general case, there can be multiple goals, adaptations and components involved. The algorithms used by Policaby to decide which adaptation instances better suit the goals will be studied in Section 3.3.

3.2.2 Adaptations

Adaptations represent operations that can be executed over the managed system, resulting in changes on the operational envelope. Examples of adaptations can be changing the BFT protocol being used, changing the BFT protocol leader, adding a replica to the system, removing a replica from the system, etc.

⟨*adaptation*⟩ ::= '`Adaptation`' name '`:`' \n ⟨*input*⟩? ⟨*requires*⟩? ⟨*impacts*⟩ ⟨*stabilization*⟩? ⟨*script*⟩?

⟨*input*⟩ ::= '`Input:`' \n ⟨*inputParam*⟩+

⟨*inputParam*⟩ ::= paramName '`:`' paramType \n

⟨*requires*⟩ ::= '`Requires:`' \n ⟨*requirement*⟩+

⟨*requirement*⟩ ::= ⟨*expr*⟩

⟨*impacts*⟩ ::= '`Impacts:`' \n ⟨*nonDetImpact*⟩+
| '`Impacts:`' \n ⟨*impact*⟩+

⟨*nonDetImpact*⟩ ::= '`[`' probability '`]:`' \n ⟨*impact*⟩+

⟨*impact*⟩ ::= ⟨*delete*⟩
| ⟨*new*⟩
| ⟨*kpiImpact*⟩

⟨*delete*⟩ ::= '`delete`' instanceVar \n

⟨*new*⟩ ::= 'new' componentName '(' ⟨*initValues*⟩ ')'

⟨*initValues*⟩ ::= ⟨*initValue*⟩
| ⟨*initValue*⟩ ',' ⟨*initValues*⟩

⟨*initValue*⟩ ::= paramName ':' value

⟨*kpiImpact*⟩ ::= kpiName '=' ⟨*expr*⟩

Each adaptation must have a name that, by good practices, should identify what the adaptation does. As mentioned in the overview, the adaptations are parameterized by component types. There are no conceptual restrictions on the number nor the types of the parameters. During the instantiation of an adaptation, each combination of inputs is generated resulting in that many adaptation instances. For instance, if an adaptation specifies two parameters *r1* and *r2* of type *Replica* and there are 4 *Replica* instances during the instantiation process, $4 * 4 = 16$ adaptation instances will be created, one for each pair of Replicas.

Some adaptations may only work in certain conditions and it is possible that not all combinations of input are valid. For that reason, adaptations can specify requirements in the form of conditions. For instance, in the example of having two parameters of type *Replica*, maybe the user needs to ensure that both parameters are distinct. In that case, she can specify a requirement (under the *Requires* construct): `r1 != r2`. This prevents having an adaptation instance in which both *Replica* parameters are the same. In the example in Listing 3.1, our adaptation assumed that the current protocol in use was Zyzzyva therefore we stated as a requirement: `protocol = Zyzzyva`. In that example we also made sure that we don't switch the leader to a replica that is already the leader: `leader != r`.

Adaptations must detail the expected changes in the operational envelop. However, as defended in (Cámara, Lopes, Garlan, and Schmerl 2016), there is uncertainty in the effects produced by an adaptation. In order to cope with that non-determinism of the predicted impacts, several branches of impacts can be defined. A branch simply aggregates a collection of impacts with its expected probability. The sum of the probabilities of all branches must be equal to 1. This allows the specification of impacts that depend on unknown variables. As an example, let's assume that changing from PBFT to Zyzzyva when the leader latency is low results in twice the throughput 90% of the cases but the remaining 10% don't affect the throughput at all. This may occur due to a variable that we are not capturing. Specifying the impacts in these non-deterministic branches, it can be stated that only 90% of the times we expect the throughput to double: `[0.9]: throughput = throughput * 2`. In Section 3.3 we will discuss how these probabilities affect the choice of which adaptation should be selected. An adaptation can only specify impacts on objects in its scope (namely, global parameters and instances passed as input). In addition, it can also specify the creation and destruction of component instances. To give a practical example, let's assume that a *Replica* component has a *cost* parameter. We can specify the creation of a new Replica with cost 2.57 as: `new Replica [cost: 2.57]`.

Frequently, after executing an adaptation, the managed system may take some time to stabilize.

During that period, the metrics gathered by the monitoring system are most likely very temporary. Decisions based on those transient values should be discouraged. For that reason, the language allows the specification on a stabilization period for each adaptation: the expected time, after executing that adaptation, that it takes for the managed system to stabilize its operation and the monitoring system data to be reliable again.

Finally, an adaptation may have an adaptation script associated. This script is written in Groovy, a dynamic programming language for the Java platform. We chose Groovy due to its synergy with the Java, the language used in Policaby's implementation. This means that Groovy scripts can access all the libraries and APIs provided by Policaby, making it easier to write adaptation scripts. For instance, Policaby provides easy to use APIs to create and remove component instances as well as multicast signed messages to all managed system's replicas. If no adaptation script is provided, the default action of executing an adaptation instance is to multicast the adaptation name and its concrete input instances to each replica of the managed system.

3.2.3 Components and Instances

As discussed above, adaptations can be parametrized by formal parameters whose types are components. Components were introduced in the language to solve two fundamental problems:

1. Specifying instantiable types whose instances can change overtime.
2. Specifying enumerated types.

One example of an instantiable type is *Replica*. The number of replicas in the system may vary overtime (as a result of adaptations that change the replica set) and each replica may have its configuration changed overtime. An example of a useful enumerated type in the context of BFT systems is the BFT protocols themselves. *Protocol* can be a type whose instances represent the specific BFT protocols. Although no changes to those instances are expected, it is useful to represent them as a type so that the user can specify a system property representing the protocol being used, for instance.

⟨*component*⟩ ::= '`Component`' name ('`:`' \n ⟨*componentParams*⟩)?

⟨*componentParams*⟩ ::= '`Params:`' \n ⟨*componentParamDef*⟩+

⟨*componentParamDef*⟩ ::= ('`monitored`')? name '`:`' type

⟨*instance*⟩ ::= '`Instance`' name '`Of`' component ('`:`' \n ⟨*instanceValues*⟩)?

⟨*instanceValues*⟩ ::= '`Values:`' \n ⟨*instanceValue*⟩+

⟨*instanceValue*⟩ ::= paramName '`:`' value \n

⟨*enum*⟩ ::= '`Enum`' enumName '`Instances`' ⟨*enumInstances*⟩

⟨*enumInstances*⟩ ::= instanceName
| instanceName ',' ⟨*enumInstances*⟩

Components must have a name and can have parameters associated. Instances are concrete occurrences of a component. Each instance has an set of values associated with its component parameters independent from the other instances, i.e. given two instances $i1$ and $i2$ from the same component C with a parameter p, $i1.p$ is an independent value from $i2.p$. Despite mentioning that components were introduced to specify instantiable and enumerated types, the language does not distinguish between them. The *Enum* construct is syntactic sugar for specifying a component with the *Enum*'s name, no parameters and creating instances with the given names.

Component parameters can be either monitored or non-monitored. A parameter is considered to be non-monitored unless stated otherwise with the keyword *monitored*. Monitored parameters have values provided by the monitoring system. For instance, the average latency of a replica to the other replicas should be a monitored parameter as its value does not depend solely on Policaby's decisions. For that reason, Policaby can only know the value of those parameters if the monitoring system provides a value for them. On the other hand, the value non-monitored parameters depend solely on Policaby's decisions. For instance, if we assume the cost of a replica is defined by the replica type (e.g. micro, small, medium, etc.) then its value is defined the moment the replica is added to the system and only changes by Policaby's order. In that case, the replica's cost does not need a value provided by the monitoring system and should be a non-monitored system. If the replicas' cost is instead determined and updated by a cloud provider each hour (as an example), Policaby has no way of knowing the current value unless it is a monitored parameter.

3.2.4 System Construct

Global parameters and configurations can be specified in a special *System* construct. The purpose of this construct is to allow the characterization of the current system configuration (for instance, the BFT protocol being used) and hold system wide metrics as the number of replicas in the system. Due to their nature, system parameters are always in scope when evaluating adaptation's requirements, adaptation's impacts or other any other expressions. This fact is used in the example given in Listing 3.1, where the adaptation changes a system parameter *leader* and uses another system parameter, namely *protocol*, in its requirements.

Besides specifying global parameters of the system, internal configuration options of Policaby can be set within this language construct. To give a couple of examples, it is possible to configure the adaptation selection algorithm (as will be discussed in Section 3.3) and the minimum period before selecting adaptations. As discussed above, adaptations can specify a stabilization period. The time specified is in the system configuration is considered to be the minimum time between selections because Policaby executes after the maximum between the configuration time and the stabilization period of the selected adaptation passes.

⟨*system*⟩ ::= ⟨*header*⟩ ⟨*systemParams*⟩
| ⟨*header*⟩ ⟨*config*⟩
| ⟨*header*⟩ ⟨*config*⟩ ⟨*params*⟩

⟨*header*⟩ ::= '`System:`' \n

⟨*systemParams*⟩ ::= '`Params:`' \n ⟨*systemParam*⟩+

⟨*systemParam*⟩ ::= name '`:`' value \n

⟨*config*⟩ ::= '`Config:`' \n ⟨*configOption*⟩+

⟨*configOption*⟩ ::= key '`:`' value \n

3.2.5 Key Performance Indicators

As surveyed in Chapter 2, configurable adaptive systems base their decisions on key performance indicators (KPIs). In the scope of Policaby, all global parameters and instance parameters are considered KPIs. There are, however, some relevant indicators that are not easily captured by those KPIs. To support with an example, suppose that each replica has an associated cost. Despite each individual cost being a KPI, the total cost of all replicas could not be retrieved. To solve that issue, compound KPIs were introduced.

⟨*kpi*⟩ ::= '`KPI`' name '`IS`' (⟨*func*⟩ | ⟨*expr*⟩)?

⟨*func*⟩ ::= ('`Sum`' | '`Avg`' | '`Min`' | '`Max`') component '`.`' param

Compound KPIs can either be an expression or an aggregation of component instance's parameters. There are four functions available to aggregate component instance's parameters, namely the sum, average, minimum and maximum of those values. Compound KPIs do not depend on any specific component instance therefore their scope can be global just like system parameters. Examples of the usage of compound KPIs can be seen in Listing 3.2. As a note, the fifth KPI in the example is just an expression: the invocation of a function that will return the bigger of its two arguments. Expressions in Policaby are very expressive and What constitutes an expression will be detailed in Section 3.2.6

```
1 KPI total_cost Is Sum Replica.cost
2 KPI max_latency Is Max Replica.latency
3 KPI min_latency Is Min Replica.latency
4 KPI avg_latency Is Avg Replica.latency
5 KPI max_latency_deviation Is MAX(max_latency - avg_latency, avg_latency -
    min_latency)
```

Listing 3.2: Example of compound KPIs.

3.2.6 Expressions

Expressions are used across the language, for instance in the specification of adaptation requirements and impacts, compound KPIs and goals. Expressions represent numeric and boolean values, however their value is always a number. The number 0 is considered to be *false* value and all other numbers are considered to be *true*. Although all non-zero numbers are considered to be true, the language guarantees that boolean expressions always have the value 1 if they are true. For instance, $(3 > 2) == 2$ is guaranteed to be zero (i.e. false) as $(3 > 2)$ would evaluate to 1 and $1 == 2$ is false. The grammar bellow described exactly what is considered an expression.

⟨*expr*⟩ ::= ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| ⟨*value*⟩
| ⟨*func*⟩

⟨*op*⟩ ::= '+' | '−' | '∗' | '/' | '%' | '^' | '&&' | '||' | '>' | '>=' | '<' | '<='
| '==' | '=' | '!=' | '<>'

⟨*value*⟩ ::= number
| name

⟨*func*⟩ ::= name '(' ⟨*args*⟩ ')'

⟨*args*⟩ ::= ⟨*expr*⟩
| ⟨*expr*⟩ ',' ⟨*args*⟩

The grammar details the available operators. The language provides the typical operators any programmer as grown to expect ($==$ and $=$ are alias, as are $!=$ and $<>$). However, all operators are binary and due to that fact, there are a couple frequent operators missing from this language, namely the boolean *not* operator and the unary minus operator. This is an unfortunate consequence of only supporting binary operators. The unary minus operator problem can be overcome by the use of $(0 - x)$, being the same as $-x$, and the not operator by the use of a function. There are several functions implemented, for instance the *NEG* function which operates as a logical *not* of its single argument. Other implemented functions include mathematical functions (e.g. trigonometric functions, rounding functions, logarithms, square root, absolute value, etc.), minimum and maximum functions (which take an unbound number of parameters and return the minimum or the maximum, respectively) and an *IF* function. The *IF* function has a behaviour similar to the conditional ternary operator found in many programming languages: it takes an expression as the first argument and if its value is true (i.e. not zero) then it returns the second argument; otherwise the third argument is returned. A few examples of valid expressions are shown in Listing 3.3

```
1 MAX(100, MIN(avg_latency, 0.6 * max_latency))
2 IF(replica.isActive, replica.cost, 0)
3 ROUND(avg_latency, 2)
```

```
4 NEG(TRUE) == NEG(NEG(FALSE))
```

Listing 3.3: Example of valid expressions.

3.2.7 Goals

Policaby's objective is to select adaptations that guide a managed system in path of accordance with its business goals. In order to that it is fundamental to capture what those business goals are. Policaby improves on the techniques used in (Rosa, Rodrigues, and Lopes 2011), by using an order list of goals able to cope with the non-determinism of adaptations.

The techniques proposed by Liliana et al. (Rosa, Rodrigues, and Lopes 2011) generalizes the typical single goal of maximizing an expression by allowing the definition of several ordered goals. Policaby extends the ordered-goal based approach of defining introducing mechanisms able to deal with the non-determinism (absent from Liliana's et al. work).

⟨*goal*⟩ ::= '`Goal`' ⟨*exact*⟩ \n
| '`Goal`' ⟨*optimization*⟩ \n

⟨*exact*⟩ ::= ⟨*expr*⟩ ⟨*confidence*⟩?

⟨*confidence*⟩ ::= '`Confidence`' probability

⟨*optimization*⟩ ::= '`Minimize`' ⟨*expr*⟩ ⟨*equivalence*⟩?
| '`Maximize`' ⟨*expr*⟩ ⟨*equivalence*⟩?

⟨*equivalence*⟩ ::= '`EquivalenceRange`' number

There are two types of goals: *exact* and *optimization*. Exact goals consist of an expression. They clearly define states of conformity and nonconformity. If evaluating the goal expression results in a false value then the state is not in conformity; otherwise it is. This type of goals are very useful to express bounds to certain KPIs. For instance we may place an upper bound on the replica's cost, **`Goal`** `total_cost <= 3`, the cost cannot be bigger than 3 Euro per hour. An important note to make is that units of the KPIs are responsibility of the user – to Policaby they are just numbers.

Optimization goals on the other hand do not define states of conformity and nonconformity but an order on those states. Which adaptation instances are selected depend on the value of all other adaptation instances being tested. For instance, if we are minimizing the cost we must first order all adaptation instances by the value of *total_cost* and only then we know which one minimizes the value. If nothing else is specified, these goals select the all the adaptation instances that maximize or minimize (depending on the goal type) the expression. Using the cost example, if the minimum value for *total_cost* is 2.2, all adaptations with $total\_cost > 2.2$ will be excluded. In certain situations this can be seen as unwanted behaviour as for instance an adaptation instance with *total_cost* equal to 2.21 would be excluded and

its value is not that worse than the minimum. For that reason, optimization goals can specify an equivalence range, a number denoting a radius in which values for the expression are considered equivalent and therefore accepted. If the equivalence range of the cost goal is 1 and the minimum value 2, then all adaptations instances that evaluate *total_cost* to a value in the interval $[2, 2+1]$ would be accepted, mitigating the problem mentioned above.

It has been mentioned that Policaby uses adaptation instances' impacts to evaluate the goals expressions, however the impacts are non-deterministic. As we've seen in Section 3.2.2, adaptations state their impacts in branches with associated probabilities. In order to cope with that non-determinism, exact goals can specify a confidence value $c \in]0, 1]$, meaning we want Policaby to guarantee that goal with probability greater than or equal to c. In order to pass an exact goal, the probability of the branches that pass that goal must be greater than or equal to c. If no confidence value is specified, it is assumed to be 1. Optimization goals on the other hand use the average of the expression value on each branch, weighted by its probability. Listing 3.4 gives some examples of goals.

```
1 Goal total_cost < 3 Confidence 0.9
2 Goal total_cost < 5
3 Goal Minimize total_cost EquivalenceRange 2
4 Goal Maximize 0.4 * throughput + 0.6 * latency
```

Listing 3.4: Example of goals.

3.2.8 Strategies

The decision algorithm of Policaby chooses a single adaptation instance to be performed. This method however prevents Policaby from detecting sequences of adaptations that may improve the managed more than a single adaptation. As a practical example, suppose there are two adaptations: (1) adding a new replica to the managed system and (2) switching the BFT protocol's leader replica. Using the constructs described so far and given that Policaby selects a single adaptation, each time Policaby executed its decision algorithm with would check whether or adding a new replica or switching the BFT protocol leader (either one or the other, exclusively) would improve the managed system, in terms of its goals. Let's assume that, in this scenario, adding a new replica do the system would violate the most important goal and, as a consequence, would not be selected. However, adding that replica and changing the leader to it would pass all goals. Despite that possibility, Policaby would fail to add the replica as it would only look for the local maximum, and adding a new replica is not the action with the most immediate return.

To overcome this problem we introduced strategies. A strategy is a tree defining possible adaptations that may be benefic to perform in sequence. In the example below the tree would degenerate in a simple sequence, but in the general case we could express several alternative adaptations to execute after adding the new replica. Below we present the grammar of the *strategy* statement.

⟨*strategy*⟩ ::= 'Strategy' name ':' \n ⟨*root*⟩ ⟨*node*⟩+

⟨*root*⟩ ::= 'root:' adaptationName '->' ⟨*edge*⟩+ \n

⟨*node*⟩ ::= nodeName ':' adaptationName '->' ⟨*edge*⟩+ \n

⟨*edge*⟩ ::= '[' ⟨*expr*⟩ ',' nodeName ']'
| 'done'

As we can see from the grammar, every strategy has a *root* node and other following nodes, connected by edges. Each node has an associated adaptation and edges have guard expressions. An edge is only considered to be a possible path if its guard expression evaluates to *true*. A possible specification of the strategy used to motivate this construct can be found in Listing 3.5.

```
1 Strategy NewReplicaThenSwitchLeader:
2     root: newReplica -> [true, leaderNode]
3     leaderNode: switchLeader -> done
```

Listing 3.5: Example of Strategy in Policaby.

Strategies do not represent binding contracts for Policaby. This means that if Policaby selects the *newReplica* adaptation due to this strategy, it is not guaranteed that the *switchLeader* adaptation would be performed next. The strategies only prevent the root adaptation from being excluded as the result of failing some goal, if the strategy itself can pass the goal (in Section 3.3.3 we will study what it means for a strategy to pass a goal). The reason supporting this decision of strategies not being binding contracts is simple: Policaby has a decision algorithm able to select the best adaptations under the current system conditions. If strategies were binding contracts, some of the power of that algorithm would be lost. In the example if, after adding a new replica, switching the leader is still the best adaptation to perform, then it will, for sure, be chosen. However, if the algorithm detects a better adaptation to be executed, then it will be chosen instead. If Policaby were to be bound to the strategies, less favourable paths could be taken.

3.3 Policaby's Engine

In addition to creating this new language for adaptation policies specification, an engine able to read a policy file written in that language and select the best adaptations based on that information was developed. This section details the implementation of the decision algorithm used by the engine to select adaptation instances and some of the protocols supporting that algorithm.

3.3.1 Decision Algorithm

Policaby's ultimate goal is to decide (and then execute) adaptations. This decision process is made in accordance with the managed system goals, as described in Section 3.2. An high level view of the decision algorithm is shown in Algorithm 1.

Algorithm 1 Policaby's Decision Algorithm.

```
1: function SELECT(Adaptations, Goals)
2:     Candidates ← INSTANTIATE(Adaptations) +[φ]
3:     Candidates ← FILTERBYREQUIREMENTS(Candidates)
4:     Candidates ← COMPUTEPREDICTIONS(Candidates)
5:     return FILTERBYGOALS(Candidates, Goals)
6: function FILTERBYGOALS(Candidates, Goals)
7:     for all goalinGoals do
8:         PassingInstances ← TEST(goal, Candidates)
9:         if |PassingInstances| = 1 then
10:            return PassingInstances
11:        if |PassingInstances| > 1 then
12:            Candidates ← PassingInstances
13:    return Candidates
```

The decision protocol, described in Algorithm 1 by function *SELECT*, starts by instantiating all adaptations into their respective adaptation instances. A special ϕ adaptation instance is added to the list of candidate adaptation instances. That instance corresponds to an adaptation that performs no action and has no impacts. Its presence guarantees that Policaby only selects an adaptation instance if it improves over not executing one at all. All adaptation instances that fail to pass their requirements are then removed from the set of candidates. After that selection, all the candidate adaptation instances have their impacts computed. This process iterates over the impact functions of each candidate, computes the expected state after execution and stores those predictions within the adaptation instance object. Predicting the impacts after validating the requirements guarantees that computational power will not be spent computing the predicted state of adaptations are not available for selection.

The remaining part of the algorithm consists in choosing which adaptations provide a better predicted state according to the goals. The goals are in the same order as specified in the policy file. Starting from the first (and therefore most relevant) goal, Policaby tests which adaptation instances pass that goal and filter out the ones which fail. If no candidate can pass a goal, then it is ignored and all candidates proceed to the next one. If a single adaptation instance can pass some goal, then it is the better one and the algorithm can terminate. If all goals are tested and there are still more than one adaptation instances in the candidates collection, then they all are considered equivalent and a deterministic function chooses one amongst them to be executed.

As Policaby is intended to execute in a BFT system, all functions must be deterministic. This means that if we execute this algorithm in different replicas, the outcome must be exactly the same. For that reason, only deterministic data structures and functions were used to implement this algorithm.

As hinted in Section 3.2.2, instantiating an adaptation is the process of associating each combination of concrete component instances with the formal input parameters of the adaptation. An adaptation with no formal parameters results in a single adaptation instance. For instance, an adaptation with two formal parameters $p1$ and $p2$ of a type T that has 5 instances results in $5 * 5 = 25$ adaptation instances. It is important to note that since $p1$ and $p2$ are of the same type, 5 of the 25 instances will have $p1 = p2$. In order to achieve only distinct pairs of T, a requirement specifying $p1 \neq p2$ should be specified. Testing

if an adaptation instance fails to meet its requirements is very straightforward. Since requirements are expressions, we can just evaluate each one, and if any requirement expression evaluates to zero (i.e. *false*), then that adaptation instance fails its requirements. Computing the the adaptation instance is the process of evaluating each expression specified on its impacts and saving the result. This prevents evaluating the same expression over and over, saving computing power. Testing whether or not an adaptation instance passes a goal is a far more complex protocol, worthy of its own section, but in order to understand it we need first to study how strategies work.

3.3.2 Strategy Instantiation Protocol

As studied in Section 3.2.8, strategies allow the user to represent sequences of adaptations in the form a tree. It was also mentioned that strategies are used to prevent the root node's adaptation instance from being excluded due to failing to pass a goal. It is therefore fundamental to understand how strategies work before detailing the goal testing protocol. The definition of a strategy is a tree whose nodes are adaptations and the edges may have associated guard conditions.

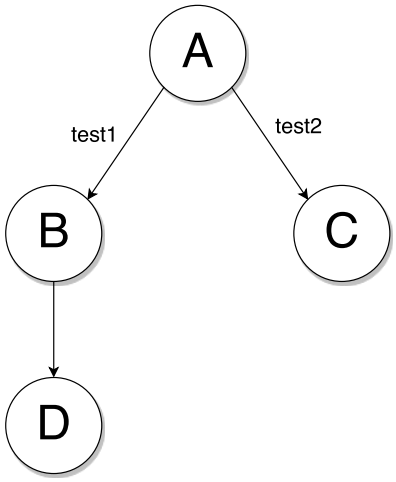

Figure 3.1: Visual representation of a Strategy S

Just like adaptations, strategies must be instantiated: its nodes are adaptations and adaptations must be instantiated, resulting in strategies needing to be instantiated as well. In order to describe the strategy instantiation protocol, the abstract strategy S represented in Figure 3.1 will be used. A, B, C and D are adaptations and B, C and D result in 3, 3 and 2 adaptation instances, respectively. Note that the number of A instances is not relevant as the root node will be associated with an instance from the $Candidates$ collection from the decision algorithm. In order to instantiate the strategy, Policaby needs to instantiate all possible paths of adaptation instances.

The resulting strategy instance can be seen in Figure 3.2, where X_i denotes instance i of adaptation X. Adaptation D had to be instantiated three times, one for each instance of B. This combinational explosion is however needed as the expected impacts of $A_0 \rightarrow B_1 \rightarrow D_1$ may differ from $A_0 \rightarrow B_2 \rightarrow D_1$, for instance. The expected impact of any adaptation instance in the strategy instance tree is computed

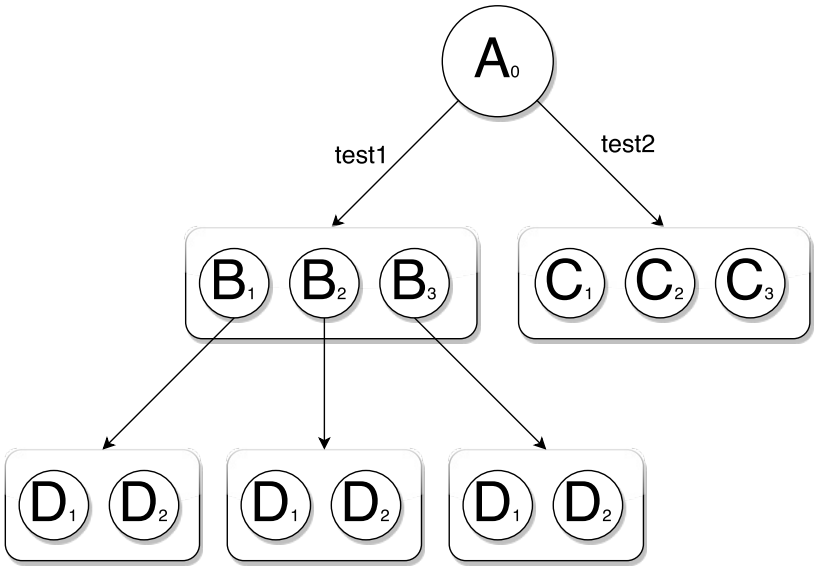

Figure 3.2: Visual representation of a Strategy S's instance.

by merging the expected state of its predecessor with the evaluation of its impact functions in the context of the predecessor predicted state. For instance, if A_0 states that some KPI k would decrease to half, B_1 that k would become $k + 10$ and D_1 that k doubles, the predicted value for k at $A_0 \rightarrow B_1 \rightarrow D_1$ would be $k = ((k/2) + 10) * 2$.

This instantiation becomes more complex due to the fact that impacts are split in branches. We need to predict each branch as many times as the number of branches of the predecessor adaptation instance and multiply their probabilities. For example, if A has two branches with probabilities 0.4 and 0.6 and B has also two branches with probabilities 0.1 and 0.9, each $B_i, i \in \{1, 2, 3\}$ will have 4 branches with probabilities $0.4 * 0.1 = 0.04$, $0.4 * 0.9 = 0.36$, $0.6 * 0.1 = 0.06$ and $0.6 * 0.9 = 0.54$. If D has 3 branches, each $D_i, i \in \{1, 2\}$ will have $4 * 3 = 12$ branches.

Strategies are the most computational intensive constructs of Policaby due to this combinatorial explosion of both adaptation instances and branches and should therefore be used conscientiously.

3.3.3 Goal Testing Protocol

The last step of the selection algorithm presented in Section 3.3.1 is to filter the candidate adaptation instances using the goals. The protocol for deciding whether an adaptation instance passes a goal depends on the type of goal: exact or optimization.

3.3.3.1 Exact Goal Testing Protocol

The protocol to decide if an adaptation instance is in accordance with an exact goal is described in Algorithm 2. Since exact goals have an associated confidence value c, Policaby needs to ensure that the adaptation instance passes satisfies the goal's expression with a probability greater than or equal

to c. In practice, an adaptation instance will pass an exact goal if the sum of the probabilities of the branches that pass the goal are greater or equal to the confidence value of the goal. A branch is said to pass a goal if evaluating the goal's expression in the context of the predicted state of that branch gives a non-zero value (i.e. true).

Algorithm 2 Exact goal testing protocol of an adaptation instance.

```
1: function InstancePassesExactGoal(Instance, Goal)
2:     sum ← 0
3:     for all branch in Branches(Instance) do
4:         if Evaluate(Expression(Goal), Predictions(branch)) ≠ 0 then
5:             sum ← sum + Probability(branch)
6:             if sum ≥ Confidence(Goal) then return true
7:     return false
```

If an adaptation instance ai fails to pass an exact goal, it is possible its strategies may still be able to pass it. Therefore, if they have not been instantiated yet, all strategies whose root is the adaptation associated with ai are instantiated with ai as the root (as described in Section 3.3.2). There are two methods available for dealing with strategies and exact goals – *Prune* and *MatchAll* – configurable using the *System* construct described in Section 3.2.4.

Using the *Prune* configuration and starting from the root node, nodes are pruned from the strategy instance tree if they (1) do not pass the goal and (2) all child nodes are recursively pruned. Basically, this method ensures that if an adaptation instance does not pass itself the goal, there is a child adaptation instance that does – meaning there is always a path from the root to a node that passes the goal. The pruned nodes are permanently removed from the strategy instance tree to ensure they are not taken into account in future goals. Using this method, Policaby assumes that a strategy passes a goal if, after pruning process, the strategy instance tree still has nodes. This is considered to be an optimistic approach to strategies as a single path to success implies the success of the strategy.

A more conservative method of dealing with strategies and exact goals is to require that all edges starting in the root node have a path to success. This corresponds to the *MatchAll* configuration. Using this configuration and starting from the root node, nodes are kept in the strategy instance tree if (1) they pass the goal or (2) after recursively applying this *MatchAll* method to child nodes there is still at least one adaptation instance associated with each edge of the strategy tree. Again, the strategy is said to pass the goal if after applying this method the strategy instance tree still has nodes. Using the strategy instance tree from Figure 3.2, at least one of B_1, B_2, B_3 and one of C_1, C_2, C_3 must be present in the tree in order to keep A_0 in the tree. Using the previous described *Prune* method, if one of $B_1, B_2, B_3, C_1, C_2, C_3$ would be kept in the tree, the strategy instance would be said to pass the goal. For that reason *MatchAll* is considered a more pessimistic approach to dealing with strategies and exact goals.

3.3.3.2 Optimization Goal Testing Protocol

Optimization goals try to maximize or minimize expressions. Unlike exact goals, whether a candidate adaptation instance passes an optimization goal depends on the predicted states of all other candidates. This happens because the maximum and minimum depend on the values of all candidates. For that reason, strategies are used from the beginning when dealing with optimization goals, instead of just being used whenever a candidate adaptation instance fails the goal. The algorithm used to decide which of the candidates pass an optimization goal is described in Algorithm 3.

Algorithm 3 Optimization Goal Testing Algorithm.

```
1: function TestOptimizationGoal(Candidates, Goal)
2:     for all candidate in Candidates do
3:         candidate.value ← Value(candidate, Goal)
4:     best ← BestValue(Candidates, Goal)
5:     return InRange(Candidates, best, Goal.EquivalenceRange)
```

Each candidate is associated with a value. How this value is calculated depends on how Policaby is configured to deal with strategies and optimization goals. These options will be explored below. Once each candidate has a value associated, the best value is determined (this obviously depends on whether it is an optimization or a minimization goal) and all candidates whose value is within the equivalence range of the best value are selected to pass the goal.

As mentioned, there are several methods to associate a value with each candidate. One method is to not use strategies at all. In that case, the value for an adaptation instance is simply given by the average of evaluating the goal's expression using each branch predictions, weighted by their probability. More formally, if an adaptation instance ai has n branches, the value of evaluating the goals' expression in the predicted state of a branch i is v_i and the probability associated with branch i is p_i, then the value attributed to ai is $\sum_{i=1}^{n} v_i * p_i$.

The remaining three methods all use strategies to compute the value that will be associated with the candidates. The value of leaf nodes of the strategy instance tree is computed using the method of not using strategies. It is important to note that edges in the strategy instance tree connect nodes to lists of nodes (this can be noted in Figure 3.2). Nodes in the same list are instances of the same adaptation. That means that if the path leading to a specific list of nodes is taken, any of the nodes in the list can be chosen by the selection algorithm. Due to this fact, we say that the value associated with a list of nodes is the best value of all nodes in the list. The remaining available methods correspond to different functions to aggregate the value of child node lists. The value of each non-leaf node i is given $Best(value_i, children_i)$, where $value_i$ is the value associated with the adaptation instance associated with node i and $children_i$ is some aggregation of the values of the node lists connected to n_i by an edge. There are two functions available to aggregate node lists values: *Best* and *Average*. The *Best* method is the more optimistic of the two, resulting in the best value amongst all nodes as the value for the root node. The *Average* method is less optimistic as by making the average of the node lists values it takes into account all strategy scenarios.

Summary

This Chapter presented Policaby, a robust adaptation manager able to use key metrics gathered by a monitoring system and a adaptation policy to decide which adaptations are most suited to guide a managed system in a path of accordance with its business goals. In order to do so, a novel adaptation policy language was developed, alongside an engine that uses those policies to guide a decision process of the best adaptations.

4 Evaluation

This Chapter provides an evaluation of Policaby. Section 4.1 makes a few comparisons with the other available languages for adaptation policy specification; Section 4.2 describes the experimental settings used to evaluate Policaby's engine and Section 4.3 details the experimental results.

4.1 Language Expressiveness

Policaby's novel language for adaptation policy specification merges and improves on the best qualities of other systems. For instance, Policaby's language is able to cope with the non-determinism associated with the execution of adaptations, a feature heavily inspired by the work of (Cámara, Lopes, Garlan, and Schmerl 2016). The work of (Rosa, Rodrigues, Lopes, Hiltunen, and Schlichting 2013) had the most expressive and easy to configure method for dealing with the managed system's business goals, so Policaby adopted a system similar to that one.

Merging those two features into a single system allowed Policaby to have novel mechanisms to express business goals that deal with non-determinism. Furthermore, the addition of fully parametrized adaptations, components and instances allow the specification of adaptations that could not be specified in the other systems. One such example is the specification of the leader change adaptation. Using the abstractions provided by (Rosa, Rodrigues, and Lopes 2011), the only way to specify a leader change adaptation would to have n adaptations, each one being *change leader to replica* n. This method however does not deal with a variable number of replicas. Similar problems would arise when trying to specify this adaptation using the methods proposed in (Cámara, Lopes, Garlan, and Schmerl 2016), since it does not support parametrized adaptations.

Policaby's language improves on the mechanisms proposed by other systems, allowing it to be more expressive and to capture a wider range of possible adaptations and goals.

4.2 Experimental Settings

Policaby's expressive mechanisms require more expensive computations than systems like Adapt. For that reason we are interested in evaluating the performance of Policaby's engine. In order to do so, the engine was executed on a single machine with 8GB of RAM and an Intel Core i7 860 (2.80GHz) CPU. This measures the time taken by a single replica, however, since the managed system must receive a

quorum of messages from Policaby, the time taken until the managed system actually performed the adaptation would be higher.

Whenever time is presented as a measure in this thesis, the presented time corresponds to an average of 100 experiences. This approach tries to reduce the impact of outliers in the experimental data.

4.3 Engine Evaluation

The main focus of this evaluation is to understand how the performance of Policaby's engine is affected by different adaptation policy files configurations. In order to do that, several configurations were used as input to Policaby and measures of the time taken to perform certain actions was collected.

The decision algorithm starts by instantiating all adaptations into their respective instances and then by computing the predicted state of each of those instances. One question that might be asked is if the ratio of adaptation instances by adaptation affects the time taken in that step. In order to answer that question, we measured the time used by Policaby to instantiate and predict the expected state using different configurations. All configurations resulted in 32000 adaptations instances but had a different number of instances per adaptation. The scenarios tested ranged from 32000 adaptations with just one instance per adaptation up to having 125 adaptations with 256 instances per adaptation. All adaptations stated 6 impact functions, divided equally between two branches.

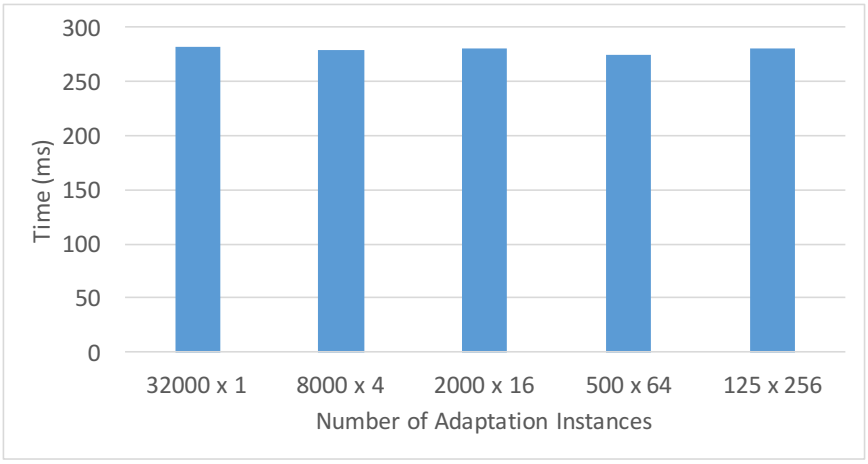

Figure 4.1: Time taken to instantiate and compute predictions of 32000 adaptations instances.

Figure 4.1 shows the experimental results of such evaluation. The ratio of adaptation instances per adaptation does not affect the time used to instantiate and compute the predicted state of the adaptation instances. For this reason, the next experience is referenced just in terms of the number of adaptation instances, ignoring the ratio of instances to adaptations.

One of the most fundamental times to measure is the time taken for the complete decision algorithm (described in Algorithm 1) to execute. This algorithm depends primarily on three variables: the goals,

the adaptation instances and the strategies available. In this experiment we are focusing on the relation between the number of goals and the number of adaptation instances, using no strategies. During this experiment, each adaptation instance had 6 impact functions distributed into two branches (with probabilities 0.4 and 0.6). All adaptation instances passed all goals. This is the worst case for the system because if all adaptation instances pass every goal, then all will be tested against every goal, maximizing the computations performed.

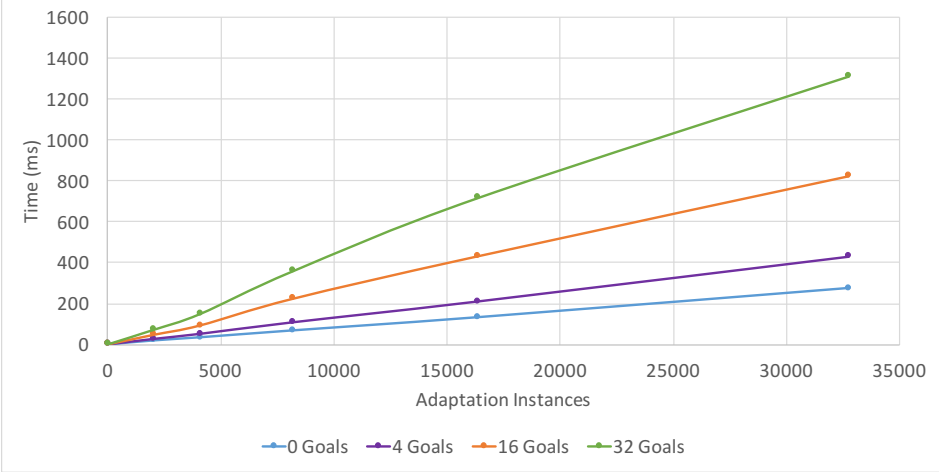

Figure 4.2: Execution time of the decision algorithm.

The data in Figure 4.2 shows that the execution time grows linearly with both the number of goals (with constant number of adaptation instances) and the number of adaptation instances (with constant number of goals). The series using 0 goals represents the time taken to instantiate and compute the predicted states. As the data shows, most of the execution time of Policaby is spent filtering the adaptation instances using goals. In this worst case scenario Policaby was able to execute the decision algorithm with 32 goals and more than 32500 adaptation instances in approximately 1.3 seconds. We think this time is very acceptable to decide which adaptation better suits the current executional environment and should not be the bottleneck to the adaptation of the managed system. A more realistic experiment is to make it so that not all adaptation instances pass all goals. We tested the 32 goals settings with approximately 32500 adaptation instances scenario with different success rates in the goals. The success rate of the goal refers to the percentage of adaptation instances that, on average, pass that goal. If the success rate is less than 100%, then the number of adaptation instances tested against each goal decreases, resulting in a lower execution time.

Figure 4.3 depicts the execution time of the slowest scenario of Figure 4.2 (32 goals and 32786 adaptation instances) using different success rates for the goals. We can see that even a very high success rate of 90% cuts the execution time into half. This scenario is much closer to a real execution of the system as in practise not all adaptation instances will pass every goal.

Strategies result in an explosion of adaptation instances to be tested. In order to visualize the increase in computation needed, we measured the number of expressions evaluated by the language with different strategies. All experiments add three adaptations A, B and C, each with two branches

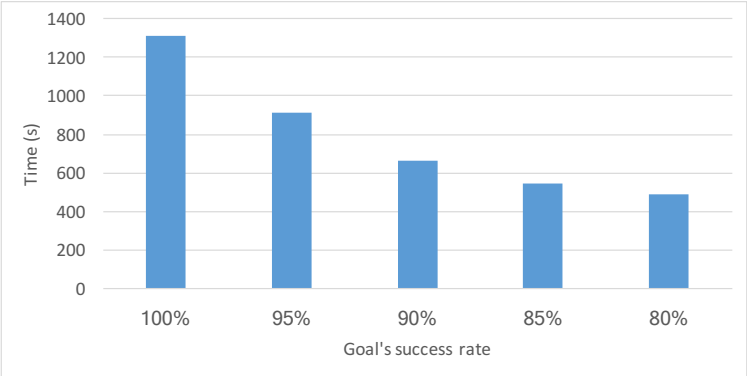

Figure 4.3: Execution time of the decision algorithm with 32768 adaptation and 32 goals with different success rates.

with a single impact function each. Each adaptation was instantiated 8 times, meaning there were 25 candidate adaptation instances in the decision algorithm. Four optimization goals with the *Best* configuration were used. The first experiment did not use strategies and is considered the base for the comparison The second experiment used a strategy $A \rightarrow B$ and the last one had a strategy $A \rightarrow B \rightarrow C$.

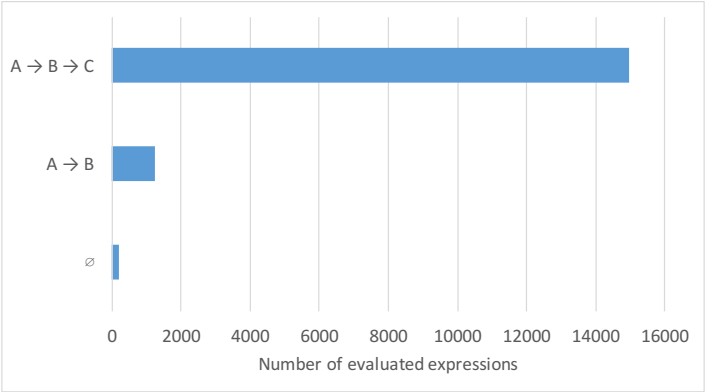

Figure 4.4: Number of evaluated expressions using different strategies.

Figure 4.4 provides a visualization of the number of expressions Policaby had to evaluate when using adaptation policies with the settings described above. As expected, strategies with multiple levels result in an explosion of the number of expressions that must be evaluated. Even a simple strategy such as $A \rightarrow B \rightarrow C$ is instantiated into 65 adaptation instances with more branches than the original ones.

Summary

This chapter introduced the experimental evaluation of Policaby, detailing the experiments conducted in order to assess its performance under different conditions. In the worst case, Policaby is able to decide the best adaptation instances from a set of over 32500 candidates, using 32 goals in approximately 1.3 seconds. More practical settings reveal that this time is greatly reduced if not all adaptation instances pass all goals (which is to be expected in the real world). Strategies are very expensive constructs and should be used mindfully.

5 Conclusions and Future Work

Despite several BFT protocols having been proposed in the literature, each one is optimized for specific operational conditions. This motivated the design of adaptive BFT systems, able to adapt to the current executional envelope. Unfortunately, the existing adaptive systems fail to provide expressive mechanisms for adaptation policy specification.

This thesis presented Policaby, a robust adaptation manager whose objective is to guide a BFT managed system in a path of accordance with its business goals. A novel language was designed to allow the specification of adaptation policies targeting Byzantine fault tolerant systems. This language provides several mechanisms such as adaptation parametrization, the ability to cope with non-determinism and goal based objectives, allowing the user to write easy to understand and expressive adaptation policies. In addition, an engine able to read an adaptation policy file in this novel language and decide which are the best adaptations under the current conditions was developed.

As future work, strategies should be revisited as they are very inefficient. For instance, it would be interesting studying a way to filter the instantiation of adaptation instances when instantiating a strategy (for example, after adding a replica, a strategy should only test switching the leader to added replica and not all replicas). The possibility of introducing parallel execution to the decision algorithm should also be studied, providing a better performance for the system. The non-determinism support in Policaby was carefully designed to be compatible with the output of Ramun (Duarte 2016). It would be interesting the connect both systems to obtain more accurate adaptation impacts.

References

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

Aublin, P.-L., R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić (2015, January). The next 700 bft protocols. *ACM Trans. Comput. Syst. 32*(4), 12:1–12:45.

Bahsoun, J.-P., R. Guerraoui, and A. Shoker (2015). Making bft protocols really adaptive. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, Hyderabad, India, pp. 904–913. IEEE Computer Society.

Bessani, A., J. a. Sousa, and E. E. P. Alchieri (2014). State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, Atlanta, GA, pp. 355–362. IEEE Computer Society.

Bradley, P. S. and O. L. Mangasarian (2000, January). k-plane clustering. *J. of Global Optimization 16*(1), 23–32.

Cámara, J., A. Lopes, D. Garlan, and B. Schmerl (2016, October). Adaptation impact and environment models for architecture-based self-adaptive systems. *Sci. Comput. Program. 127*(C), 50–75.

Castro, M. and B. Liskov (1999). A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Cambridge, MA, USA.

Castro, M. and B. Liskov (2002, November). Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*(4), 398–461.

Cheng, S.-W. and D. Garlan (2012, December). Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw. 85*(12), 2860–2875.

Clement, A., E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti (2009). Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, Boston, Massachusetts, pp. 153–168. USENIX Association.

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*, OSDI '06, Seattle, Washington, pp. 177–190. USENIX Association.

Duarte, F. (2016, November). Learning adaptation models under non-determinism. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa.

Kotla, R., L. Alvisi, M. Dahlin, A. Clement, and E. Wong (2010, January). Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst. 27*(4), 7:1–7:39.

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

Lamport, L., R. Shostak, and M. Pease (1982, July). The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*(3), 382–401.

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

Pasadinhas, M., D. Porto, A. Lopes, and L. Rodrigues (2017). Adaptação guiada por políticas de sistemas tolerantes a faltas bizatinas. In *Actas do Nono Simpósio de Informática (INForum)*, Aveiro, Portugal.

Rosa, L., L. Rodrigues, and A. Lopes (2011, November). Goal-oriented self-management of in-memory distributed data grid platforms. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2011)*, Athens, Greece. (short paper).

Rosa, L., L. Rodrigues, A. Lopes, M. A. Hiltunen, and R. Schlichting (2013, March). Self-management of adaptable component-based applications. *IEEE Trans. Softw. Eng. 39*(3), 403–421.

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

Smola, A. J. and B. Schölkopf (2004, August). A tutorial on support vector regression. *Statistics and Computing 14*(3), 199–222.

van Renesse, R. and F. B. Schneider (2004). Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, San Francisco, CA, pp. 7–7. USENIX Association.