Adapting Byzantine Fault Tolerant Systems

Miguel Neves Pasadinhas
miguel.pasadinhas@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Malicious attacks, software bugs 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. In this report we study the mechanisms and policies that permit to build adaptive BFT systems, able to adjust the used algorithms in response to changes in their operational envelope.

1 Introduction

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[1]. 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[2] 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 [1], 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[3–7].

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[6] 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[8]. 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[9]. Still, as we will discuss later in this report, 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. In this work we are interested in studying policy-based adaptation of BFT systems. We are interested in two complementary aspects: first we aim at identifying policies that may be relevant in practice and experimentally assess the benefits of those policies; second we are interested in identifying the right abstractions to express those policies, such that they can be easy to write and easy to understand, without requiring a deep understanding of the internals of all components that execute in the system.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of building adaptive BFT systems. In particular, we are interested in building an *adaptation manager* that can execute a flexible set of policies to drive the adaptation of a target BFT state machine replication system (*the managed system*). The adaptation manager collects information from a monitoring tool, and runs an adaptation policy that drives the re-configuration of the managed system. The adaptation manager must be, itself, tolerant to Byzantine faults thus it will be implemented as a distributed set of independent replicas that will run a BFT-consensus protocol to coordinate their actions. More precisely:

Goals: Our work is to build a robust and flexible adaptation manager for BFT systems.

Our work builds on an early prototype of architecture to support the adaptation of BFT system named ByTAM[10]. The adaptation manager provided in ByTAM only supports a small set of policies, in the form of Event-Condition-Action policies. Our goal is to extend the expressiveness of the policies that can be supported by the adaptation manager.

The project will produce the following expected results.

Expected results: The work will produce i) a specification of the adaptation manager; ii) an BFT implementation of the manager, iii) a set of policies for the dynamic adaptation of BFT systems, iv) an extensive experimental evaluation using a fully operational prototype.

3 Related Work

In this section we give an overview of previous work addressing adaptation Byzantine Fault Tolerant systems. We start by describing some of the existing BFT protocols, to motivate the need for changing the protocol according to the conditions of execution environment. Next we survey previous adaptive BFT systems. Then we overview BFT-SMaRt, one of the few open source libraries supporting BFT state machine replication, and ByTAM, an early prototype on which we will build on. For completeness, at the end of the section, we also survey a number of systems that, although not targeting BFT systems specifically, can provide useful input to our work.

3.1 Byzantine Fault Tolerance

In this subsection we give an overview of four different BFT protocols, namely: the original protocol by Lamport, Shostak, and Pease[1], PBFT[3], Zyzzyva[6] and Aardvark[7]. We start by the original formulation provided in [1] even if these protocols only work in synchronous systems, then we address PBFT because 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.

3.1.1 Byzantine Generals Problem The term Byzantine Fault Tolerance was introduced by Lamport, Shostak, and Pease[1]. 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 the 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.

3.1.2 PBFT The Practical Byzantine Fault Tolerant protocol[3] (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 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 [3] and a detailed proof is given in [11].

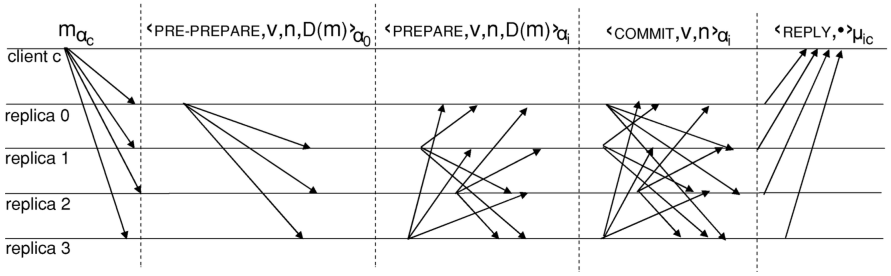

Fig. 1: PBFT failure free run. replica 0 is the primary replica. Taken from [3].

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 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.

3.1.3 Zyzzyva Zyzzyva[6] 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, 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[6].

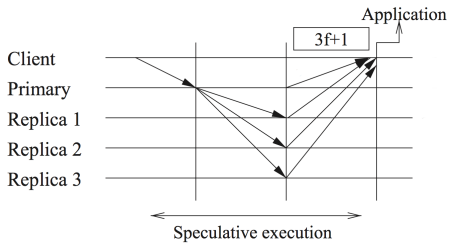

Fig. 2: Message pattern of Zyzzyva's fast case. Taken from [6].

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 case, depicted in Figure 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.

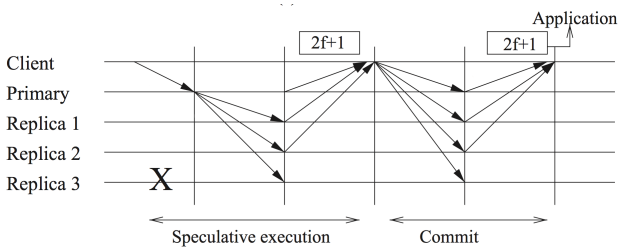

Fig. 3: Message pattern of Zyzzyva's two-phase case. Taken from [6].

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.

3.1.4 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[7] 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.

3.2 Adaptable Byzantine Fault Tolerant Systems

As we have seen from the three practical protocols discussed above, each one 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*. We will discuss each of these 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[10].
- Replace a module: allowing for changing the implementations of a given module. One example is switching the BFT protocol that the system is using[8, 9].

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 we will study in this report 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[10], there is contention[8] or even some protocol will have at least a 10% better performance than the current one[9]. The techniques proposed in [12] 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. This issue is being addressed by other members of our research group and is considered to be out of the scope of this report. Changes to the replica set will probably 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[8] or (2) having both running at the same time during a transition period until the older is definitely removed[13]. 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.

3.3 The Next 700 BFT Protocols

The observation that different protocols perform better in different conditions has motivated the authors of [8] 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. In the remainder of this report we will cover several systems and techniques focused on the adaptation of systems.

3.3.1 Abstract In their work, the authors of [8] 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.

3.3.2 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.

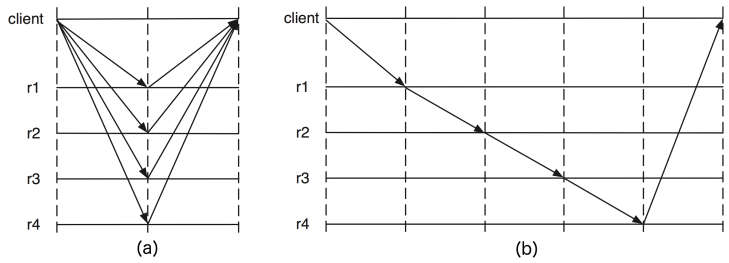

Fig. 4: Message pattern of (a) Quorum and (b) Chain. Taken from [8].

Quorum is a protocol with a very simple message pattern, depicted in Figure 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 [14] (which only tolerated crash), adapted to tolerate Byzantine faults. As can be seen in Figure 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.

3.3.3 Policies The authors have experimented to build adaptive BFT protocols based on the instances above. Two variants of the system have been de-

veloped, 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.

3.3.4 Discussion Abstract allows the creation 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.

3.4 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.

We will now explain how Adapt tries to predict the KPI values used in the evaluation process. The KPI values are computed in runtime using prediction mechanisms, in this case Support Vector Machines for Regression[15]. 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.

We mentioned above that 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. They 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.

We conclude this analysis of Adapt 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 policies. We can specify policies 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 we can specify when they trigger, 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.

3.5 BFT-SMaRt

BFT-SMaRt[16] is an open-source library for creating BFT SMR systems and will be used in the development of our system. 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.

As hinted above, BFT-SMaRt has a reconfiguration module. Currently, it only supports adding and removing replicas. Other members of our research group are studying ways of allowing BFT-SMaRt to make other adaptations, for instance changing the BFT protocol. That, combined with the fact that there is significant knowledge of this system in our research group, makes BFT-SMaRt a good choice for implementing our system.

3.5.1 ByTAM ByTAM[10] is an early prototype of a BFT Adaptation Manager built using BFT-SMaRt. As we mentioned before in this report, our work will build on this system. 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. As mentioned in Section 2, our goal is to improve the expressiveness of the policies, so that the writer can focus on the business goals of the system.

3.6 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.

3.6.1 Impact Models Under Uncertainty Cámara et al. defend in [17] 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[18].

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 3.6.2 we will study machine learning techniques that help mitigate this problem.

3.6.2 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[19]. 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[20] algorithm
5. The result is processed and written to a text file

The resulting text file is compliant with the language proposed in [17]. 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.

3.6.3 Rank-based Policies and Adaptations Rosa et al. 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).

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.

4 Architecture

We plan to build a robust Adaptation Manager, able to choose adaptations based on user specified policies. As a requirement for our system to work properly, a Monitoring System should exist and provide the Adaptation Manager with insightful metrics about the Managed System, i.e. the target system of the adaptations. Both, the Monitoring System and the Managed System must have APIs compatible with our system.

Since we are focused on the adaptation of BFT systems, our system will be itself tolerant to Byzantine faults. In order to achieve that, we will leverage BFT-SMaRt[16] and our system will be a service implemented with that library. As a result, our Adaptation Manager will inherit BFT-SMaRt's architecture and machinery.

4.1 Overview

The proposed architecture for the Adaptation Manager service is depicted in Figure 5. The *Metrics Storage Service* is responsible for storing metric values sent

by the Monitoring System. It is, therefore, the bridge between our system and the Monitoring System. On the other hand, the *Adaptation Coordinator* makes the bridge between our system and the Managed System. It will be responsible for communicating with the Managed System and coordinate the adaptations. The *Policy Manager* is responsible for managing the policies life-cycle and execution, and the *Model Reviser* has the task of refining the policies models. We will now look further into each of those modules.

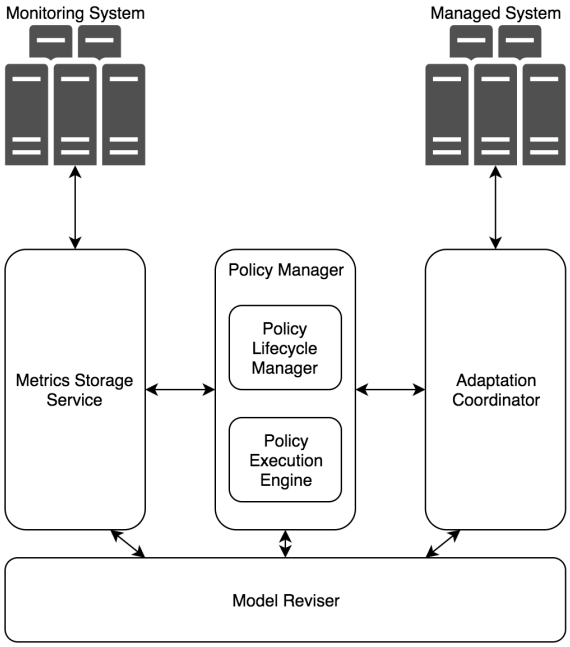

Fig. 5: Architecture of the proposed Adaptation Manager and its interactions with the Monitoring and Managed Systems.

The Metrics Storage Service can be seen as the entry point of the service, as the other modules will execute in reaction to the arrival of new metric values (which we will refer to as just *values*, from now on). When new values arrive from the Monitoring System, they are stored in the Metrics Storage Service. The Policy Manager and Model Reviser modules are notified and can act upon those values. To prevent the Metrics Storage Service from growing indefinitely, older values should be removed. This process will be triggered by the Model Reviser and discussed below.

The Policy Manager is responsible for handling the policies and is divided in two big modules. The *Policy Lifecycle Manager* provides the system with the ability of having a dynamic set of policies as it allows the installation and removal of policies. This provides the system with great flexibility as an operator

can change the policies in runtime, without the need of restarting the system. The *Policy Execution Engine* is responsible for enforcing the policies. Given a set of policies, this module will strive to satisfy them by ordering adaptations. This module is highly dependent on the type of policies supported by our system – it is very different to satisfy Event-Condition-Action rules compared to enforcing a limit on some key metric, for instance. This module must implement an algorithm able to cope with all the features of our policy specification language. Although an exact specification of such language is still not defined, we present some key features we strive for in Section 4.2.

Once the Policy Manager orders an adaptation, the *Adaptation Coordinator* is responsible for communicating the decision to the Managed System. It is this module's job to prepare the messages and mediate the communication with the Managed System. After sending the necessary messages, this module will expect responses from the Managed System so it can be sure the adaptation succeeded. In order to handle any arbitrary adaptation, it is possible to associate an *user defined function* to each adaptation. That function will be called by the Adaptation Coordinator when the associated adaptation is triggered, allowing for a complex coordination process.

With the three modules described above, the Adaptation Manager is able to work properly. The Model Reviser works orthogonally to the other modules and uses the values stored in the Metrics Storage Service to refine the prediction models used by the policies. Its responsibility is highly dependent on the models used by the policy specification language. To illustrate a possible responsibility of this module, lets assume the policies were specified using the techniques proposed by Rosa et al. in [12]. In that language, the models correspond to the impact functions associated with the adaptations. The Model Reviser module would therefore use the knowledge gathered in the Metrics Storage Service to refine the impact functions – it is notified when an adaptation is made and uses the changes in the metrics, before and after the adaptation, to refine the impact functions. With this method, the impact functions would become more accurate as they are refined using real data, allowing the system to decide better adaptations. When this module uses data from the Metrics Storage Service and detects it will not longer use that data for other revisions, it orders that service to delete the unneeded data, so it does not grow indefinitely.

4.2 Policies Specification

Although the exact specification of the language used to express policies is still not defined, there are some key characteristics we would like it to have. Most of these characteristics are based on the systems and techniques discussed in Section 3 and are as follow:

- *Focus on business goals.* We want the language to allow the user to focus on the business goals intended for the system.
- *Event-Condition-Action.* In some cases, the user may want to decide exactly what the system should do, therefore supporting this kind of policies is an

interesting feature. For instance, a system expert may want to decide what adaptation should be executed when a replica is faulty instead of trusting a complex decision process to decide what adaptation should be done.
- *Boundaries for key metrics.* Allowing the user to specify ranges of acceptable values for some key metrics. The system should then try to keep this metrics in the given range.
- *Optimization policies.* The objective is to inform the system of what is the optimal value for a given metric, so it can find adaptations that make the metric approach the given value.
- *Graceful Degradation.* Specifying an order on the policies allows the Adaptation Manager to ensure maximum effort in fulfilling those more important business goals.
- *Non-determinism.* As Cámara et al. discussed[17], sometimes we do not have all the variables to predict the impact of an adaptation. Being able to cope with that uncertainty would be a desired feature for our language. One example might by moving a replica to a different machine (while keeping its IP) – if that replica happens to be the leader the system will need to perform a view change; otherwise if there are less than f faulty replicas, the system can cope with that transition period without a view change. With the work of [19], new non-determinism cases can be learnt even if the operator does not think of them.

Most of the desired characteristics for our language are covered by the language proposed by Rosa et al. in [12], making it a good candidate to be extended in order to support non-determinism and event-condition-action policies.

5 Evaluation

In order to evaluate our system, we will need a managed system able to adapt and a monitoring system. Fortunately, other members of our research group are developing such systems. The managed system will be able to perform replica set reconfigurations (as it will also be developed using BFT-SMaRt) and protocol switching adaptations. The monitoring system will provide several metrics including latency, throughput, number of concurrent clients, request size, faults, among others.

To evaluate our proposal we will implement the Adaptation Manager described in Section 4 and develop a set of policies. We will then use the managed system and monitoring system described above. The managed system will be subjected to heterogeneous execution environments (including faulty ones). This allows our Adaptation Manager to have trigger various adaptations, based on what it thinks is the best for the current conditions. We will measure:

- The time needed between receiving the data and deciding on a adaptation. This will determine how fast the managed system can react to changes in the execution envelope.

– The quality of the chosen adaptations. This will be measured by evaluating changes in key metrics, such as latency and throughput. It will reflect the quality of the method used for choosing adaptations, the quality of the model reviser and the quality of the developed policies set.

As we are interested in a robust Adaptation Manager we will also test how it performs under faults – either faults of own replicas or faults in the managed system or monitoring system.

6 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

7 Conclusions

Since the Byzantine Generals Problem was introduced, several BFT Protocols were proposed. PBFT was one of the first protocols exhibiting an acceptable performance operating on asynchronous systems. After it was proposed, several variations emerged, each one with better performance in specific system conditions. Due to the fact that each protocol is tuned to be optimal in different system conditions, it is fundamental to be able to adapt BFT systems.

The techniques introduced in [8] allowed for the creation of Aliph – a BFT system able to switch the BFT protocol in runtime. Unfortunatly, Aliph only switched protocol based on predefined static conditions. Adapt[9] further explored the ideas introduced in Aliph, being able to switch the BFT protocol based on the predicted performance of other protocols under the current system conditions. Despite that, Adapt lacks a good and expressive policy specification system.

There are several techniques used in non-BFT adaptive systems that may improve the quality of a BFT Adaptation Manager. Cámara et al. proposed a language to specify policies, based on impact functions, able to handle non-determinism[17]. Using machine learning techniques, the impact functions can be refined, allowing for more accurate models of the adaptations[19]. Rosa et al. proposed a language to specify policies as an ordered set of *goals*. This allows the system to degrade gracefully as it will ensure maximum effort satisfying goals in the given order.

We propose the construction of a BFT Adaptation Manager. Our work will build on an early prototype[10] built using BFT-SMaRt[16] – a BFT library to create BFT systems. We aim at improving the policies expressiveness by taking inspiration in the techniques discussed in this report.

Acknowledgments We are grateful to Carlos Carvalho, Bernardo Palma, Daniel Porto and Manuel Bravo for the fruitful discussions and comments during the preparation of this report. This work has been 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.

References

1. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3) (July 1982) 382–401
2. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
3. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4) (November 2002) 398–461
4. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. SIGOPS Oper. Syst. Rev. **39**(5) (October 2005) 59–74
5. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: 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, USENIX Association (2006) 177–190
6. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. ACM Trans. Comput. Syst. **27**(4) (January 2010) 7:1–7:39
7. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: 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, USENIX Association (2009) 153–168
8. Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 bft protocols. ACM Trans. Comput. Syst. **32**(4) (January 2015) 12:1–12:45
9. Bahsoun, J.P., Guerraoui, R., Shoker, A.: Making bft protocols really adaptive. In: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium. IPDPS '15, Hyderabad, India, IEEE Computer Society (2015) 904–913
10. Sabino, F.: Bytam: a byzantine fault tolerant adaptation manager. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (September 2016)
11. Castro, M., Liskov, B.: A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Cambridge, MA, USA (1999)
12. Rosa, L., Rodrigues, L., Lopes, A., Hiltunen, M.A., Schlichting, R.: Self-management of adaptable component-based applications. IEEE Trans. Softw. Eng. **39**(3) (March 2013) 403–421
13. Mocito, J., Rodrigues, L.: Run-time switching between total order algorithms. In: Proceedings of the Euro-Par 2006. LNCS, Dresden, Germany, Springer-Verlag (August 2006) 582–591

14. van Renesse, R., Schneider, F.B.: 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, USENIX Association (2004) 7–7
15. Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. Statistics and Computing **14**(3) (August 2004) 199–222
16. Bessani, A., Sousa, J.a., Alchieri, E.E.P.: 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, IEEE Computer Society (2014) 355–362
17. Cámara, J., Lopes, A., Garlan, D., Schmerl, B.: Adaptation impact and environment models for architecture-based self-adaptive systems. Sci. Comput. Program. **127**(C) (October 2016) 50–75
18. Cheng, S.W., Garlan, D.: Stitch: A language for architecture-based self-adaptation. J. Syst. Softw. **85**(12) (December 2012) 2860–2875
19. Duarte, F.: Learning adaptation models under non-determinism. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (November 2016)
20. Bradley, P.S., Mangasarian, O.L.: k-plane clustering. J. of Global Optimization **16**(1) (January 2000) 23–32