Policy-Based Adaptation of Byzantine Fault Tolerant Systems

(extended abstract of the MSc dissertation)

Miguel Neves Pasadinhas
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***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.

I. 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], [4], [5], [6], [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, 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. 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.

II. Related Work

After the initial BFT protocols initially proposed in [1], several more efficient and able to work in a wider range of conditions were derived [3], [6], [7], [4]. Unfortunatly, each protocol is designed to show better performance under certain execution conditions, reveeling poor performance outside those conditions. To the best of our knowledge, there is no single BFT protocol able to outperform all other under every conditions. For that reason, adaptive BFT systems able to change their behaviour according to the execution conditions have been designed.

One of the first systems able to switch the BFT protocol due to variations in the executional envelope was Aliph [8]. Unfortunatly, this system has a single adaptation policy, consisting in replacing the BFT protocol whenever certain conditions are met. The available protocols are ordered in a ring and the protocol switching always follows that order, i.e. when a protocol is deactivated, the one that follows it in the ring is activated. The conditions defining when each protocol should be deactivated are hardcoded in the system and capture the programmers knowledge of the protocols performance. There is no support for defining new policies in Aliph. Another downside is that Aliph is a monolithic system, meaning that the system itself decided its own adaptations.

Adapt[9] is a BFT system with a three sub-systems architecture, namely: the target system which will be adapted (BFTS), an event system (ES) responsible for monitoring the BFTS and a quality control system (QCS) that uses the data gathered by the ES and decided adaptations. This architecture is inspired by the control sequence MAPE-K [10], frequently used in non-monolithic adaptive systems. Despite the BFTS being BFT, neither the ES nor the QCS are. Like Alpih, the only available adaptations are protocol switching. Adapt however, introduced an intermediate step to decide which protocol should be the next to execute. Adapt uses machine learning techniques to predict the value of key metrics if each protocol was executing under the current executional conditions. The user must specify a weight to each of those key metrics and the protocol with the best weighted average will be chosen for execution. Although slightly configurable, the adaptation policy of Adapt is still very simple. It is hard to map the business goals of the BFTS into a simple weight vector, making this configuration mechanism very hard to tune.

ByTAM [11] was a prototype that followed a three sub-system architecture, similar to Adapt, with a managed system, a monitoring system (MS) and an adaptation manager (AM). Unlike Adapt, all three subsystems were BFT. ByTAM allows the specification of generic adaptations, being the only of the three not limited to protocol switching. Adaptations are specified in event-condition-action form and the policy goals are implicitly encoded in those rules. Adaptations require the user to implement a Java interface, using Java code. Although very flexible, this approach lacks good abstracts to facilitate writing adaptation policies, therefore loosing expressiveness.

More expressive and flexible techniques to specify adaptation policies can be found in the literature. For instance, Stitch [12], [13] allows the specification of tactics (adaptive primitives with actions and expected impacts) that cope with the non-determinism associated with adaptations. The non-determinism is captured by specifying the tactics impacts using discrete time Markov chains. Tactics can be grouped in strategies, tree structures that define complex operations that can be executed on the system. The best strategies are selected by the system, recurring at the expect impacts of their tactics in key metrics. Key metrics have a weight associated and the system tries to maximize the weighted average of the key metrics. Liliana et al. [14] proposed a novel way of specifying the managed system's goals. Instead of maximizing a single expression that typically involves key metrics and weights associated with them, their system allows the specification of several, ordered goals. Goals function as filters, and the possible adaptations are tested (in order) against every goal, until the best one is selected. If an adaptation fails to pass a goal, it is removed from the set of possible adaptations to be executed. Although these two languages provide expressive mechanisms to specify adaptation policies, they lack crucial abstracts for BFT adaptation. As an example, none of these languages is able to specify an adaptation that selects the fittest replica to be the leader of the BFT protocol.

III. Policaby

This section presents Policaby, a reliable adaptation manager with a novel language for adaptation policy specification. Section III-A presents the system architecture assumed by Policaby, Section III-B details the novel adaptation policy specification language made for Policaby and finally Section III-C describes important protocols and algorithms used by Policaby during execution.

A. 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 Policaby's 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. For that reason, the metrics' values service should be versioned and, if all Policaby's replicas use the same version *id*, then all can make the same decision (given that Policaby's decision algorithm is deterministic, which it is). We now reduced the problem of all replicas using the same data to 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.

B. 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.

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 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 III-C.

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.

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 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 [12], 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 III-C 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) 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.

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.

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 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 III-C) 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.

5) Key Performance Indicators: As surveyed in Chapter II, 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.

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 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 III-B6

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

Listing 2. Example of compound KPIs.

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 bellow described exactly what is considered an expression.

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. Listing 3 provides examples of valid expressions.

```
MAX(100, MIN(avg_latency, 0.6 * max_latency))
```

```
IF(replica.isActive, replica.cost, 0)
ROUND(avg_latency, 2)
NEG(TRUE) == NEG(NEG(FALSE))
```

Listing 3. Example of valid expressions.

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 [14], by using an order list of goals able to cope with the non-determinism of adaptations.

The techniques proposed by Liliana et al. [14] 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).

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 III-B2, 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 4 provides examples of goals.

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

Listing 4. Example of goals.

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.

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 5.

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

Listing 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 III-C3 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.

C. 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.

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 III-B. An high level view of the decision algorithm is as follows: (1) Adaptations are instantiated into a $Candidates$ collections; (2) a special ϕ adaptation instance is added to the $Candidates$ collection; (3) $Candidates$ are filtered based on their requirements; (4) the expected states of the $Candidates$ is computed; (5) the goals are used as filters to select the best candidates.

In more detail, the decision protocol 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 III-B2, 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.

2) Strategy Instantiation Protocol: As studied in Section III-B8, 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.

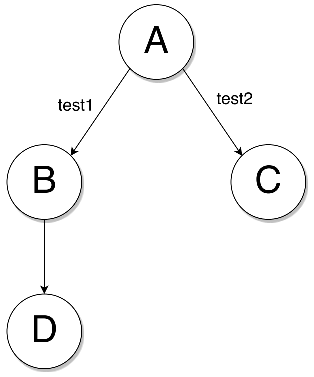

Figure 1. Visual representation of a Strategy S

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 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.

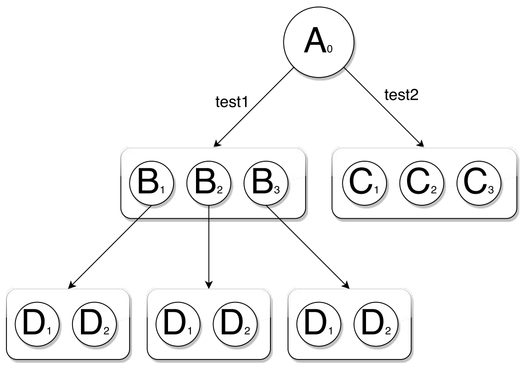

Figure 2. Visual representation of a Strategy S's instance.

The resulting strategy instance can be seen in Figure 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 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) Goal Testing Protocol: The last step of the selection algorithm presented in Section III-C1 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.

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).

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 III-C2). There are two methods available for dealing with strategies and exact goals – *Prune* and *MatchAll* – configurable using the *System* construct described in Section III-B4.

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 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.

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 as follows: (1) each candidate is associated with a value; (2) the best value amongst all candidates is found; (3) the candidate with the best value and all whose values are within the equivalence range of the best value are selected.

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 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.

IV. Evaluation

In order to evaluate Policaby's performance, several experiments with different settings were conducted. All experiments times and metrics were measured in a single Policaby replica, with 8GB of RAM and an Intel Core i7 860 (2.80GHz) CPU. Each presented time correspond to the average of 100 experiences, as a tentative of diluting outliers.

The time taken by Policaby to run the decision algorithm was measured. The number of adaptation instances varied with the experiences, and so did the number of goals. All adaptation instances had two branches of impacts with 3 impacts functions each. All adaptation instances passed every goal. This scenario corresponds to the worst case for Policaby, as every adaptation instance must be tested against every goal.

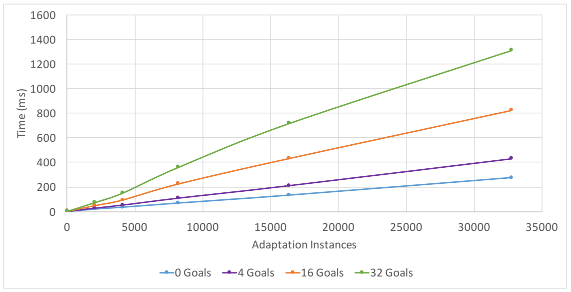

Figure 3. Policaby's execution time under different settings.

As we can see from Figure 3, the execution time grows linearly with the increase of goals (given constant number adaptation instance) and adaptation instances (given constant number of goals). We can see that Policaby took approximately 1.3 seconds to execute its decision algorithm with more than 32500 adaptation instances and 32 goals. A scenario closer to reality would be if not all adaptations passed all goals. For that we tested the 32 goals with approximately 32500 adaptation instances with different success rates of the goals. The success rate defines the average percentage of adaptation instances that pass the goals.

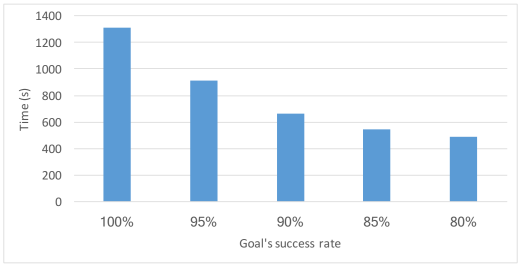

Figure 4. Policaby's execution with different goal success rates.

Figure 4 depicts those experiments. With a still very high success rate of 90%, the execution time dropped to half. Strategies are the most expensive constructs of the language. To access their impact on the number of computations Policaby needs to perform, we measured the number of expressions evaluated under different scenarios. All three scenarios had three adaptations, A, B and C that were instantiated into 8 instances each. The system had 4 maximization goals, using the *Best* method for dealing with strategies. The first test case had no strategy and 210 expressions were computed. Adding a strategy $A \rightarrow B$ increased the number of computed expressions to 1246. Adding an extra edge to the strategy, becoming $A \rightarrow B \rightarrow C$, increased the number of computed expressions even further to 14966. From these results we conclude that strategies should be used mindfully.

V. Conclusions

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.

Acknowledgments

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. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely Carlos Carvalho, Bernardo Palma and Daniel Porto.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[3] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1095809.1095817

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: A hybrid quorum protocol for byzantine fault tolerance," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 177–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298473

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1658357.1658358

[7] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 153–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558988

[8] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 12:1–12:45, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2658994

[9] J.-P. Bahsoun, R. Guerraoui, and A. Shoker, "Making bft protocols really adaptive," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15. Hyderabad, India: IEEE Computer Society, 2015, pp. 904–913. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.21

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[11] F. Sabino, "Bytam: a byzantine fault tolerant adaptation manager," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Sep. 2016.

[12] J. Cámara, A. Lopes, D. Garlan, and B. Schmerl, "Adaptation impact and environment models for architecture-based self-adaptive systems," *Sci. Comput. Program.*, vol. 127, no. C, pp. 50–75, Oct. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2015.12.006

[13] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *J. Syst. Softw.*, vol. 85, no. 12, pp. 2860–2875, Dec. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2012.02.060

[14] L. Rosa, L. Rodrigues, and A. Lopes, "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, Nov. 2011, (short paper).