ByTAM: a Byzantine Fault Tolerant Adaptation Manager

(extended abstract of the MSc dissertation)

Frederico Miguel Reis Sabino
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—Previous systems that support the dynamic adaptation of Byzantine Fault Tolerant (BFT) protocols have important limitations such as: lack of robustness (dependence on central and trustworthy components); low flexibility (constraining the adaptation strategies); and lack of extensibility (making it impossible to add new policies regarding the current service in operations). To overcome these gaps, this dissertation proposes a generic BFT adaptation manager which can be used to execute different adaptation policies. This manager is independent of the target service which needs to be adapted dynamically and is, by itself, also tolerant to Byzantine Faults. The manager has been implemented using BFT-SMaRt[1], a well known open-source library. The experimental evaluation of the resulting prototype illustrates that it can be used to effectively increase the performance of a target BFT service in different operational conditions.**

I. Introduction

State machine replication with Byzantine Fault Tolerant (BFT) protocols is a technique that allows building robust services, capable of working correctly, even in the presence of a minority of arbitrary faults (accidental or malicious), caused by external attacks or intrusions. Essentially, these protocols implement a solution of *consensus*, a fundamental problem in distributed systems that has been extensively studied in literature[1], [2], [3].

BFT is required to operate correctly in face of faults but it must also exhibit good performance. In fact, the performance of the protocol also has a strong impact on the final user experience. Most BFT protocols are optimized to achieve the best performance on the most common operational conditions (eg. local networks or WAN, sporadic failure events, etc.). As a result, these protocols may be penalized when operating outside the expected conditions. For instance, Zyzzyva[4] when compared with PBFT [5], performs better when the threat level (failures or presence of Byzantine processes) is low. However, Zyzzyva's complex strategy to recover from failures ends up being less efficient than PBFT in scenarios where failures are more frequent[6].

Furthermore, the factors that affect the performance of the protocol such as the number of replicas, the size of the clients' requests or the threat level, have values that may change overtime. Thus, it becomes relevant to use techniques that allow the dynamic adaptation of the parameters and the algorithms used by the BFT services. Some protocols, such as Aliph[7] and ADAPT[8] already provide some support towards this goal, allowing a protocol change if a failure occurs or if a certain condition is not met, combining strategies of different BFT algorithms. Still, these solutions are not flexible regarding the adaptation mechanisms they support, the policies which trigger the adaptation, the adaptation strategy (change of protocol, execution cancellation), and the degree of robustness (centralized components or non-BFT). Despite the merit of these pioneer works, it is still necessary to make significant progress to derive practical solutions, which are simultaneously robust and efficient.

In this dissertation, we present ByTAM, a generic adaptation manager which allows the dynamic adaptation of a given target BFT protocol. ByTAM supports custom policies which can be dynamically selected, allowing it to execute any adaptation policy which can be expressed using rules of the type "condition-event-action". ByTAM is tolerant to Byzantine faults, where the adaptation manager and the associated monitor infrastructure operate independently of the service being replicated and adapted. The resulting architecture, which allows the fine tuning of the parameters of the target BFT protocol, requires few modifications to the protocols being adapted. This feature eases the incremental transition of legacy systems (non adaptable) to new systems with dynamic adaptation capabilities. Furthermore, ByTAM is capable of dealing with issues such as failures and asynchrony which, when unhanded, could prevent the different replicas of the adaptation manager to make consistent decisions. ByTAM is an open source project which is integrated with BFT-SMaRt[9], a library used for the development of BFT systems.

On the following sections of the extended abstract of the dissertation we present ByTAM as follows: in Section II, we address the related work. In Section III we present a global overview of ByTAM, highlighting the architecture components and their main functions. The experimental evaluation is presented on Section IV. Finally, Section V contains the conclusions for this work.

II. Related Work

In 1978, Lamport L. depicted an algorithm that introduced the concept of SMR under a distributed environment [10]. The algorithm described an extension that provided total

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

ordering. The algorithm however produced arbitrary results if it was not coherent with the decision of the system's users. A solution for this arbitrary problem is by using synchronized clocks.

Additionally, the algorithm assumed that processors never failed and that all the messages were correctly delivered. Lamport then created a real-time algorithm [11] that assumed upper bounds on message delays and that correct processes had their clocks synchronized. It is one of the first algorithms that described the idea of arbitrary faults described in The Byzantine Generals Problem [3]. However, these solutions were designed for synchronous systems where known time bounds are in place. Therefore, we cannot assume that they work on an asynchronous environment like the Internet.

Castro and Liskov, introduced the first BFT protocol that is safe under partially-synchronous systems – PBFT [5]. This protocol started the search for further optimizations where we can have a cheap solution which is both reliable and fast. Protocols such as Zyzzyva[4] and Aardvark[12] further improved on initial solutions but always with a compromise. There was no perfect protocol for a system where the operational conditions change over time – once we had a system running a particular protocol, it was impossible to replace it without shutting down the system and perform a manual reconfiguration.

In *Guerraoui et al.*[7] the authors presented a new abstraction to reduce the cost of developing BFT protocols named *Abstract*, and introduced the property of *Abortability*, which allows the composition of different BFT instances and allows to alternate correctly between the execution of these instances. In particular, the authors focused on the necessary mechanisms to replace different BFT protocols. Using previous algorithms as instances, the authors built a system named *Aliph* which combines three protocols: *Quorum*[7], *Chain*[7], and PBFT[5]. Since the emphasis of the work was not on the adaptation policies, *Aliph* uses simple criteria to start a reconfiguration, such as the occurrence of a fault (these criteria are hard-coded in the system). *Aliph* initially executes *Quorum* which offers a better performance in fault-free scenarios and, when a fault occurs, it is replaced by a *Chain* instance which is more robust but also more complex. When any other fault occurs, another adaptation is performed (this time to the PBFT instance). Therefore, reconfigurations are always executed following a sequence statically defined, namely $Quorum \rightarrow Chain \rightarrow PBFT$. Finally, after a quarantine period, *Aliph* tries to switch back to the *Quorum* instance.

ADAPT[8] is a protocol which also explores the concept of *Abortability*, although in a more flexible manner, introducing an evaluation stage to elect the next active BFT protocol. ADAPT reacts to the changes in the environment using machine learning techniques to guide the selection process. In this way, adaptations are selected based on metrics designated *Impact Factors*, in can occur even in the absence of faults (different from *Aliph*). Essentially, ADAPT is composed by three different subsystems: the *BFT System* (BFTS), composed by various BFT protocols that need to support the property of *Abortability*; the *Event System* (ES) which is responsible for monitoring the system and retrieve values for the different impact factors; the *Quality Control System* (QCS) which is responsible for the analysis of the values retrieved by the ES and start an adaption when necessary.

Although the usage of machine learning is an interesting contribution, the solution proposed in ADAPT leaves opens three aspects which are addressed by ByTAM: a robust implementation of the QCS; a robust infrastructure of the ES; and an easy way to create adaptation policies. Firstly, although the authors indicate the possibility of developing a BFT QCS, in the system proposed, the decision of making an adaptation is taken by only one replica (the primary one). Thereafter, the BFTS must trust the decision of the primary QCS; if the primary QCS is Byzantine this may compromise the correction of the service. Furthermore, despite the fact that ADAPT modules are installed at all the replicas, the system centralizes the QCS operations at a single machine. Also, the presented ES is quite simple and a more robust solution is postponed by the authors for future work. Finally, the system only supports policies which are based on *impact functions* (which capture the effect of each adaptation) and the o way to interpret these functions is hard-coded on the system, which constrains the possibility of implementing different policies.

In addition to the issues above, the choice followed by the authors of collapsing all these components on the same replicas (BFTS, ES, QCS) may affect negatively the robustness of the system. For instance, when a failure occurs on the ES of one replica and on BFTS of another replica (considering $f = 1$). This does not happen in ByTAM, where each system component operates independently and may even have distinct replication levels, as it will be described in Section III-F (the choice of running the different services on the same replica is up to the administrator).

It is also important to emphasize that, despite the large number of research projects on BFT protocols, many of them are not open source. In this work, we chose BFT-SMaRt[9], which is a BFT replicated state machine written in Java. BFT-SMaRt is a modular system which executes a BFT protocol similar to PBFT[5] and includes modules for state transfer and replica reconfiguration. In BFT-SMaRt, the process of reconfiguration is coordinated by an external component, central and trustworthy (which is one of the limitations that ByTAM tries to overcome). BFT-SMaRt provides a programming interface which is simple and extensible and its implementation follows a modular approach.ByTAM uses BFT-SMaRt in two ways: i) as an automatic adaptation management system, and ii) as the managed target system.

III. ByTAM

In this section we present the system model, the architecture, and a general perspective on how the different

components of ByTAM operate to perform adaptations in a coherent way.

A. System Model

We assume a Byzantine failure model on which processes that fail can behave in an arbitrarily manner [3] and the existence of a strong adversary, with the ability to coordinate failed processes to compromise the replicated system. However, we assume that this adversary cannot violate known cryptographic techniques such as MACs, encryption, and digital signatures. Furthermore, we assume that at most f replicas of each component may fail (from either the adaptation manager, sensors or the managed system). Finally we assume an asynchronous network where eventually synchronous intervals occur; in those intervals, messages are delivered in time and the protocol makes progress.

B. Architecture

ByTAM has two subsystems which operate independently: the Monitoring System and the Adaptation Manager. Both communicate with the Managed System, as it is shown in Figure 1(a). The Managed System is the service that is provided to the end user; it provides information on the operational conditions and receives adaptations, working as a client of ByTAM. An example of a possible managed system is DepSpace[13], a fault tolerant tuple space. The Managed System must satisfy two properties: it must have some kind of reconfiguration mechanism (eg. view change protocol or support *Abortability*) and be monitorable (exporting performance and operational metrics). The Managed System operates independently from the remaining system components so, faults that prevent communication with ByTAM (network partitioning, asynchrony) may cause delays in the adaptation process. Additionally, the Managed System can have a different failure model from ByTAM. However, to achieve a globally robust solution, in this dissertation the Managed System is also Byzantine fault tolerant. On the following sections, we will describe the Monitoring System and the Adaptation Manager, presenting their main functionalities.

C. Monitoring System

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

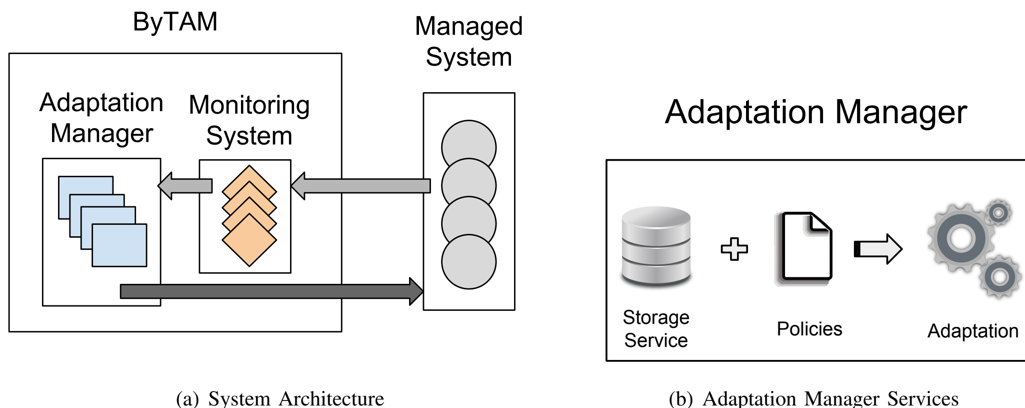
As a result, the sensors produce a linear sequence of captured values identified by tuples which, in addition to the collected data, include a unique identifier of the replica/sensor and a sequence number. These tuples are stored in the Adaptation Manager. However, individual values are not used directly.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

D. Adaptation Manager

The Adaptation Manager is responsible for processing the data obtained from the Monitoring System, register and evaluate the adaptation policies, and start the adaptation process. The Adaptation Manager has three services, as illustrated in Figure 1(b): storage, policy management, and adaptation coordinator. Essentially, the Adaptation Manager processes the data that comes from the sensors and computes quality metrics. These metrics are then evaluated by policies that were previously installed. These policies analyze the metrics obtained and decide if a reconfiguration action is needed. Like the Monitoring System, the Adaptation Manager is a Byzantine fault tolerant replicated service. It is important to remark that the messages exchanged between the Monitoring System and the Adaptation Manager are digitally signed, therefore non-authenticated sensor messages are discarded.

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

The policy manager: controls the life cycle of the adaptation policies. More specifically, it allows listing, installation/removal of policies, associate/disassociate and activate/deactivate policies of a specific Managed System.



(a) System Architecture (b) Adaptation Manager Services

Figure 1. ByTAM: Services and architecture

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

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

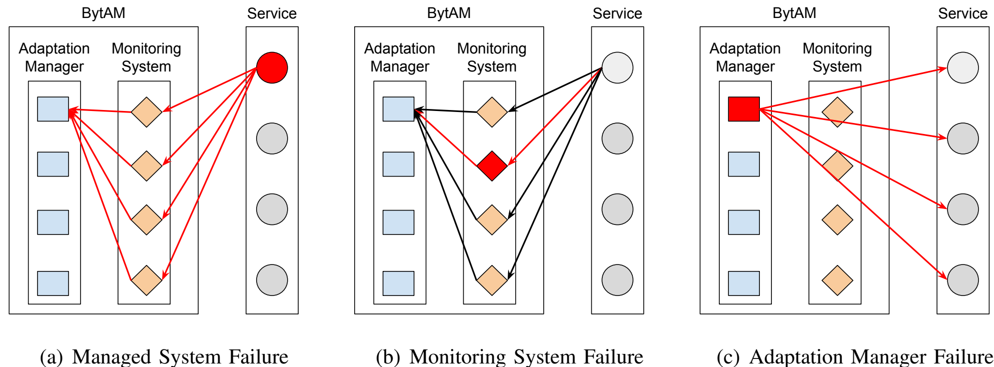
E. Performing an Adaptation

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

F. Dealing with Byzantine Components

As it was previously explained, each ByTAM component operates independently, communicating exclusively through message exchange. This eases both the analysis of the system behavior when faults occur and the mechanisms that avoid the propagation of these failures, as it is illustrated in Figure 2. In the following paragraphs we discuss the effects of the occurrence of faults on different components (namely the Managed System, the Monitoring System and the Adaptation Manager).

When one of the replicas of the Managed System exhibits Byzantine behavior, the Monitoring System readings can obtain arbitrary values or fail silently. In the example provided in Figure 2(a), the Managed System replica can emit intentionally different values for each sensor in order to force the activation of a specific policy and consequently reduce the system availability during the reconfiguration interval. To avoid the propagation of arbitrary values produced by a Byzantine replica of the Managed System, each Monitoring System replica must read the same indicator across all Managed System replicas and perform a fault tolerant voting on the values read, before propagating the reading result to the Adaptation Manager. However, most replicas of the Managed System will send coherent values to the



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

Figure 2. Messages from failed replicas on different components ($f = 1$)

Monitoring System replicas for which all the correct replicas will end up sending a correct reading to the Adaptation Manager.

When a Monitoring System component fails, as it is illustrated in Figure 2(b), it may either not provide the readings to the Adaptation Manager or, change arbitrarily the values read and send contradictory values to different Adaptation Manager replicas. The propagation of contradictory values is filtered by the consensus execution (when the Adaptation Manager receives the values). A reading omission is treated in the same way as an incorrect value. Finally, incorrect values which are received by the Adaptation Manager are eliminated in the voting process (which uses the values received by other Monitoring System replicas): the value of the reading is either greater/smaller than the values produced by other replicas or it is within the interval defined by the values received by two correct replicas. On the first scenario, the deterministic function deletes the extreme values in the voting process. On the second scenario, if the value is within the interval defined by two correct replicas, then it is also a correct value. This strategy is similar to what is used in other contexts, such as in [14].

Finally, when an Adaptation Manager replica fails (Figure 2(c)), it can either opt to not execute the policy or trigger an incorrect adaptation at an inappropriate time. However, these behaviors will be masked by the remaining Adaptation Manager replicas. It is important to recall that every Adaptation Manager replica receives the same sequence of values sent by the Monitoring System, as described earlier. In this way, each correct Adaptation Manager replica individually applies the policies and, if needed, trigger an adaptation. Since the policies are deterministic, every correct replica of the Adaptation Manager will trigger the same adaptation. Hence, even if one of the Adaptation Manager replicas cuts communication with the Managed System, the remaining Adaptation Manager replicas will end up sending the reconfiguration command, triggering the adaptation process. Even if a Byzantine replica sends an incorrect command to the Managed System, it will not start the adaptation immediately. Instead, the Managed System waits for a quorum of $f+1$ equal reconfiguration commands from different replicas to perform the adaptation.

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

G. *Adaptation Policies*

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

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

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

The development of new policies goes through the implementation of a *Policy* interface which is provided by the

Adaptation Manager. This interface provides two methods: *trigger()* and *execute()*.

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

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

IV. EVALUATION

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

A. Experimental Setup

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

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

B. The evaluated Adaptation

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

C. Results

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

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

The results are shown in Figures 3 and 4, with an increasing number of clients performing requests on the replicated service. As it is shown in Figure 3, the configuration which has the lowest latency is the configuration where the policy

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

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

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

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

Listing 1. Example of an Adaptation Policy for ByTAM.

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

We have also registered how long it took for the system to perform some reconfiguration tasks. We used the policy described in Listing 1 so the values retrieved are from reconfiguration tasks which are relevant for that policy. More specifically, we measured how long it took to i) create a new configuration for the target system. This includes retrieving relevant sensor values registered by the Adaptation Manager, create a new configuration using the Managed System reconfiguration API, and digitally signing the message; ii) reconfigure the checkpoint frequency of the Managed System (with 4 and 7 active replicas); iii) reconfigure the resilience of the system – reducing the number of active replicas from "$n = 7$" to "$n = 4$"; iv) reconfigure both the resilience and the checkpoint frequency of the Managed System.

As for the number of consensus instances performed by the Adaptation Manager, since each value from the the Monitoring System is totally ordered, we have one consensus execution for each sensor value. So, the number of consensus executions is between $n - f$ and n (where n is the number of replicas of the Monitoring System). The Adaptation Manager also sends an ADAPT message to all the other Adaptation Manager replicas when a given replica wants to run the currently active policy (as it is described in Section III-D). Since these messages are also totally ordered, a consensus instance is executed for every correct Adaptation Manager replica which is between $n - f$ and n.

V. Conclusions

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

Acknowledgments

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

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer, 2011.

[2] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

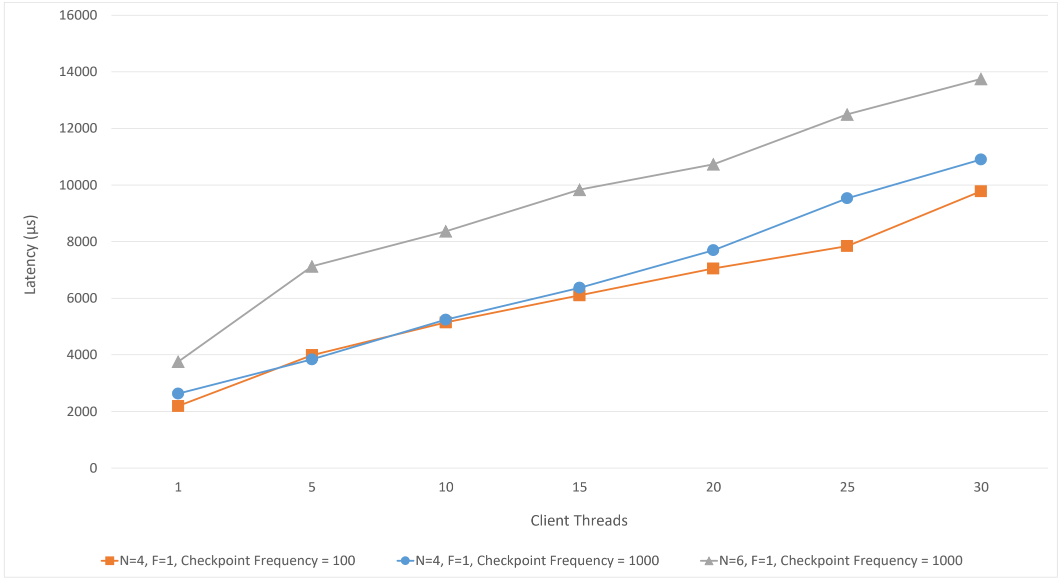


Figure 3. Latency of the various configurations

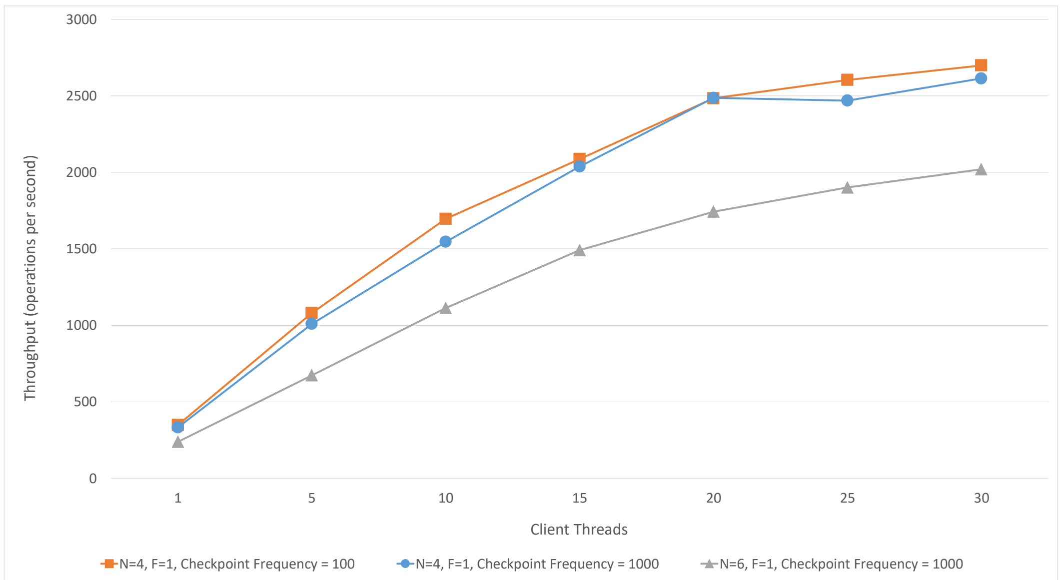


Figure 4. Throughput of the various configurations

[3] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 45–58, 2007.

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

[6] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, 2008, pp. 189–204.

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

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

[9] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with bft-smart," in *Proceedings of the 44th*

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

Table I
EXECUTION TIME FOR RECONFIGURATION TASKS DESCRIBED IN LISTING 1

Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014, pp. 355–362.

[10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] ——, “The implementation of reliable distributed multiprocess systems,” *Computer Networks (1976)*, vol. 2, no. 2, pp. 95–114, 1978.

[12] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, “Making byzantine fault tolerant systems tolerate byzantine faults.” in *NSDI*, vol. 9, 2009, pp. 153–168.

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

[14] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, “Reaching approximate agreement in the presence of faults,” *J. ACM*, vol. 33, no. 3, pp. 499–516, May 1986. [Online]. Available: http://doi.acm.org/10.1145/5925.5931