Dynamic Adaptation of Byzantine Fault Tolerant Protocols

(extended abstract of the MSc dissertation)

Carlos Eduardo Alves Carvalho
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—The problem of distributed consensus in the presence of Byzantine faults has received particular attention in recent decades. Today a variety of solution to this problem exist, each optimized for particular execution conditions. Given that, in most cases, real systems operate under dynamic conditions, it is important to develop mechanisms that allow the algorithms to be adapted at runtime or to switch between different algorithms so that is possible to optimize the system to the current conditions. The problem of dynamic adaptation of consensus algorithms is not new, but the literature is scarce for the Byzantine case and there is no comprehensive comparison of existing solutions. This work has two complementary objectives. First, it studies how the different dynamic adaptation techniques proposed for the crash failure model can be applied in the presence of Byzantine faults. Second, it presents a comparative study of the performance of these switching algorithms in practice. For that purpose, we have implemented the switching algorithms in a common software framework, based on the open source BFT-SMaRt package. Using this common framework we have performed an extensive evaluation that offers useful insights on the practical effects of different mechanisms used to support the run-time switching among Byzantine protocols.**

I. Introduction

State Machine Replication (SMR)[1] is one of the fundamental techniques for providing fault tolerance. At its core, this technique uses a distributed consensus algorithm so that all the replicas can agree in the order in which they should process the requests. This work focuses mainly on the case where one intends to use SMR to tolerate Byzantine faults (BFT). Among the systems that have been proposed to accomplish BFT state machine replication we highlight PBFT[2], Aardvark[3], and Zyzzyva[4]. Each of these systems operates better under certain conditions, and worse in others, with none surpassing all others in all situations, as shown in [5]. Zyzzyva performs better when no faults occur and the network is stable. On the other hand, when faults frequently occur, Aardvark operates better than the rest, sacrificing performance in the fault-free case. In addition, the performance of the PBFT is less sensitive to the increased size of the messages exchanged when compared to Zyzzyva. These differences motivate the interest of switching among different algorithms, or to dynamically adapt a given algorithm.

Therefore, in this work we are interested in the study of mechanisms that allow to adapt, or to replace, in runtime, a consensus algorithm for another. This is relevant since most of the practical applications of SMR are subject to variations of their execution environment, from changes in the load imposed by clients to variations on the network performance. Furthermore, as there is no one-size-fits-all algorithm solution for a range of extended operating conditions [5], the only way to ensure a good performance in face of a variable envelope is to perform dynamic adaptation.

The problem of dynamic adaptation of consensus algorithms is not new and has been well studied for the crash fault model (e.g. [6], [7], [8]). However, the literature is scarce for the Byzantine case and, in fact, several of the previously proposed mechanisms may fail in face of Byzantine faults and need to be modified to operate in such a scenario. Even among algorithms developed taking Byzantine faults into account there is, as far as we know, no work to compare their performance. In this way, those who seek to support dynamic adaptation while tolerating Byzantine faults do not have at their disposal concrete data in order to choose the adaptation technique that best suits the characteristics and objectives of the target system.

Thus, this work has two main goals. First, it studies how the different dynamic adaptation techniques that were previously proposed for the crash failure model can be adapted to work in the presence of Byzantine faults. Second, it presents a comparative study of the performance of these switching algorithms in practice. For that purpose, we have implemented the switching algorithms in a common software framework, based on the open source BFT-SMaRt library[9]. Using this common framework we have performed an extensive evaluation that offers useful insights on the practical effects of different mechanisms used to support the run-time switching among Byzantine algorithms.

II. Related Work

There have been proposed multiple techniques for dynamically adapting consensus protocols in the literature. We group these adaptive protocols into three categories: those using a single *reconfigurable consensus protocol*, those using multiple consensus protocols either *as black-boxes*, or relying on *stoppable consensus protocols*.

Using a reconfigurable consensus algorithm. One of the most straightforward ways to reconfigure a system is to develop an adaptation aware monolithic protocol, which

already incorporates all the behaviours that can be activated at runtime. In this case, the dynamic adaptation simply consists of modifying one or more configuration parameters of the monolithic implementation. Nevertheless, since consensus protocols involve multiple processes, it is necessary to coordinate the reconfiguration of the different replicas in order to ensure that they always operate in compatible configurations (typically in the same configuration). Lamport et. al. described how this technique can be carried out safely [10]. A similar technique is also used in ByTAM [7], an earlier attempt at producing a reconfigurable version of BFT-SMaRt. One limitation of this approach is that it requires all desirable behaviours be supported by a single protocol, that quickly becomes extremely complex maintain, and whose correction is difficult to ensure.

Using multiple consensus algorithms. A more modular solution consists of implementing the different behaviours using independent protocols and then having mechanisms to switch among these protocols. We designate the component that allows switching among two or more protocols a *switcher*.

The most straightforward solution is to consider consensus protocols as black-boxes. The advantage of this solution is that any protocol could be integrated without major effort, as requires no adaptive features from the consensus protocol. Unfortunately, there is no direct way for a switcher to identify whether a given protocol is already quiescent, which presents a challenge. Therefore, the switchers in the different replicas have to coordinate explicitly. An approach on this is described by Mocito et. al. in [7], nevertheless, it assumes a perfect failure detector. Bortnikov et. al. [11] presented a CFT implementation of this technique.

Switching can be facilitated if the protocols export an interface that allows the switcher to deactivate the protocol, placing it in a quiescent state[10]; in the literature, these protocols are designated as stoppable[12]. In this case, after the deactivation has been requested by the switcher, the various replicas of the protocol coordinate with each other to ensure that new messages are no longer processed. Note that there may be a gap in the communication flow during the reconfiguration, since deactivating a protocol is not instantaneous and requires coordination among the various replicas. Furthermore, not all available Byzantine consensus protocols have support for their deactivation, which limits the coverage of this approach.

Aublin et. al. presented an special type of stoppable protocols [13]. They propose algorithms, which, given a set conditions—e.g. a specific failure occurs, or the network is getting overloaded—autonomously stop. Using these algorithms, the authors propose Abstract, a BFT systems development framework that allows to deactivate one algorithm and replace it with another one when environment conditions change. When an algorithm is deactivated, it responds to all requests with a proof of termination, its operations history and an indication of the next algorithm to be activated. The client is then responsible for forwarding his request, together with the received information to the new algorithm to be used. This continuous approach brings a gap in communication due to the need for retransmission of requests. By not using a dedicated switcher, one needs to transfer more data over the network (the history) in order to ensure the correction of the system as a whole. Another disadvantage of Abstract is that it only supports a fixed set of hard-coded policies to decide which reconfigurations are meaningful.

As an optimization for adaptive techniques relying on multiple protocols, Mocito et. al. [7] and Bortnikov et. al. [11] propose to have both protocols (the one being used and the protocol to which we want to switch to) concurrently working, such that the impact of reconfiguring is unnoticeable. Thus, the switcher would send all the messages in parallel to both protocols during a reconfiguration. Of course, a disadvantage of this strategy is that it leads to a significant increase in bandwidth usage during reconfiguration since all messages are ordered by the two protocols, so it can only be applied in systems where the network does not present a bottleneck.

III. Byzantine Protocol Adaptation

In this section we discuss the operation of several protocol switching strategies in the presence of Byzantine faults. We classified this strategies in three different categories: i) *adaptation using a reconfigurable algorithm*, ii) *adaptation using algorithms as black boxes*, and iii) *adaptation using stoppable algorithms.* matching the categories described in §II. If the switching algorithms have not been originally proposed for this model, we discuss the changes that are required to efficiently apply them to BFT protocols. We also present some new optimizations that have not been previously suggested in the literature.

A. System Model

We assume the Byzantine fault model, where failing processes may exhibit arbitrary behaviour, including colluding to attack the system in a coordinated fashion. Nevertheless, we consider an adversary with limited computational resources, and without the possibility to break the cryptographic primitives used by the protocols. Finally, a partially synchronous network is assumed, in which arbitrary asynchrony periods may exist, but there is always a period of synchrony in which the system can make progress; in the moments of synchronization, the transmitted messages are delivered to the recipient within a bounded time interval.

We assume that in the system there are N replicas with the ability to instantiate several Byzantine protocols. Clients of the replicated service do not interact with these instances directly, but rather through a component called switcher, which is responsible for forwarding these requests to one or more of these protocols (making dynamic adaptation transparent to clients). For resource-efficiency, each switcher can be co-located with the respective replica instance. We also assume that there is an external component to the system, called the adaptation manager, that decides which

protocol (or configuration) is going to be used at any moment. When it is necessary to adapt, the adaptation manager sends an order to all the switches, in order to start the adaptation process. This adaptation commands are ordered and identified with a monotonically increasing id. The implementation of the adaptation manager is orthogonal to this work, being described in [14]; typically the manager itself is also replicated and each switcher only initiates switching when it receives an identical command from a quorum of adaptation manager replicas.

B. Using Reconfigurable Algorithms

As seen previously in §II, when using a reconfigurable algorithm, the main challenge is to coordinate all the replicas to use the same configuration. One of the simpler ways of doing it, as suggested in [10], is to use the consensus algorithm itself to define the logical instant at which the reconfiguration takes effect. For this, the reconfiguration algorithm must support pre-defined reconfiguration requests, which are, as application requests, submitted for consensus. Only when the replicas reach a consensus on a reconfiguration command, the reconfiguration is applied, ensuring a mild transition.

This adaptation strategy is generic and can be used assuming any fault model, including BFT. The correction of the switching algorithm depends exclusively on the properties of the underlying consensus algorithm. Therefore, if we reconfigure a Byzantine fault tolerant protocol, the reconfiguration itself also tolerates this type of faults. This technique is potentially the most efficient, especially since the protocol can be implemented in order to prioritize reconfiguration requests, causing these requests to be ordered before other messages are queued to be ordered.

Unfortunately, efficiently implementing reconfigurable consensus algorithms is significantly hard. One of the problems, apart from the obvious complexity of implementing such complex algorithms, is that each of the behaviours integrated in the algorithm may possibly require different optimizations that may be in conflict; e.g. an optimization that works for a protocol A, makes protocol B inefficient. We detail our experience building such reconfigurable algorithms in §IV.

C. Using Algorithms as a Black-Box

Adaptive strategies that consider consensus algorithms as black-boxes use control messages, namely *markers*, to coordinate the reconfiguration. Let us assume that an application requests a reconfiguration from protocol A to protocol B. In order to start the switching, each switcher (or a subset of them) sends a marker via protocol A. When the first marker is delivered (by the consensus protocol A), switches start submitting request to protocol B instead to protocol A. Consequently, from that point in time, switchers can only deliver messages received through protocol B. Unfortunately, due to the asynchrony of the system, a switcher can see several of the messages sent by itself to protocol A ordered after a marker, in which case it is obliged to re-send these messages to protocol B. In this way, there can be not only a gap during the reconfiguration, but also an increase in network usage due to the need to re-send data messages to a different protocol.

An additional difficulty to this approach in the presence of Byzantine faults is to verify that the marker corresponds to a configuration that was actually requested by the adaptation manager, to prevent a Byzantine switcher from inducing undesirable reconfigurations. We have considered two techniques for obtaining this guarantee. The first is to include in the marker a proof of veracity of the configuration request; this consists in having the reconfiguration command signed by a quorum of adaptation managers. A second solution would be to start the switching only after receiving $f + 1$ markers from different switcher replicas. This last option would eliminate the need to send the proof (since waiting for $f+1$ markers, we are sure that one of them was sent by a correct process), but would delay the switching process. For this reason, in our prototype we use the first solution.

D. Using Stoppable Algorithms

Adaptive strategies that use stoppable consensus algorithms have the potential of simplifying the reconfiguration protocol at switchers. When a deactivation is requested, these algorithms ensure that no request is ordered after the stop command. Therefore, the switch does not have to implement—as for the black-box strategies— a protocol to cope with requests that have been already submitted but not delivered before the marker.

Let us describe in more detail the steps followed to reconfigure from a protocol A to B. Before the reconfiguration is initiated by the manager, the switcher sends messages to protocol A; when the reconfiguration is triggered, the *switcher* stops submitting messages and requests the deactivation of protocol A: the flow of messages is then temporarily stopped, while the *switcher* waits for protocol A to be in a quiescent state; after obtaining confirmation of deactivation of protocol A, the *switcher* resumes sending the messages, now using protocol B. Note that there may be a gap in the communication flow during the reconfiguration, since deactivating a protocol is not instantaneous and requires coordination among the various replicas. Furthermore, not all available Byzantine consensus protocols have support for their deactivation, which limits the coverage of this approach.

In the Byzantine case, presents the following problem, which arises from most Byzantine consensus protocols, based on the election of a leading process: consider the case in process p_i is the leader of protocol B but it is the last process to be informed that A is in the quiescent state. In this case, the remaining processes, may have already switched to Protocol B, and can erroneously assume that p_i is faulty in face of the absence of the leader's activity (when in fact p_i is correct but blocked while awaiting the deactivation of A). This can start a leader-switching processes in Protocol B even before its operation is initiated in full, which can harm system's performance. For this reason, in a Byzantine

fault tolerant system, it may be preferable for a switcher replica to immediately send messages in Protocol B, even before making sure that protocol A is quiescent. Note that messages delivered by B will have to be quarantined until A is quiescent. In order to avoid delivering the same message to the application more than once, messages that have been ordered by A in the past should be discarded from the message list ordered by B and only then the remaining messages ordered by B may be delivered.

E. Optimizing with Parallelization

We have seen previously that one way to reduce the gap that may occur during switching is to send all messages to both protocols, A and B, during the reconfiguration process. The way to define the logical moment in which the switching takes place can be the same described above. However, when switching is done, messages not ordered by A will already be ordered by B and ready to be delivered.

Systems that use this technique, such as [7] and [11], assume that when the switching process starts protocols A and B are already instantiated in all replicas and that no process uses these protocols unnecessarily. In the Byzantine setting, there is a risk that a Byzantine process will start transmitting in protocol B even when no switch operation has been requested by the adaptation manager, which will lead to wasted resources and open the door for denial of service attacks. To mitigate this problem, we developed the following strategy. When a process sends a first message in protocol B, it must add to that message a proof that the switch to B has been requested. As before, this test consists of the reconfiguration command signed by a quorum of replication managers. A process that receives a message for protocol B for the first time, it only activates protocol B if the proof is valid. Otherwise, it discards the message and issues an accusation against its sender.

Although it seems an optimization, it is not clear if in practise the system will perform better when using this technique as it is resource-intensive and can lead to starvation.

IV. BFT-SMaRt Instantiation

In order to carry out an experimental evaluation, require to support a comparative analysis of the performance in practice of the various techniques mentioned above, we have implemented the different algorithms in a common software framework, namely, the BFT-SMaRt platform[9]. BFT-SMaRt is an open-source software package that enables the execution of a replicated state machine service that is tolerant to Byzantine faults. We have chosen this framework for implementing the switchers since it is one of the best performing and actively supported BFT SMR implementations. Unfortunately, the BFT-SMaRt offers a single consensus algorithm, which is a variant of the algorithm described in [15] for the Mod-SMaRt framework[16]. In order to illustrate the switching among different algorithms, we have also developed an additional algorithm for BFT-SMaRt. Namely, we have developed an implementation of the Fast Byzantine Consensus[17], which we call Fast-SMaRt. In addition, we added to the BFT-SMaRt a switcher module, which mediates the interaction among clients and the supported consensus algorithms. The switcher is prepared to receive reconfiguration commands from a replicated adaptation manager, whose development is being done by other elements of our team[14].

In addition to the switcher and the new consensus algorithm mentioned above, we have developed other extensions to BFT-SMaRt that are relevant to our experiments. In particular, we have developed stoppable versions of the Mod-SMaRt and Fast-SMaRt algorithms so that we can test switching between algorithms with support for deactivation, as well as the possibility to reconfigure the leader of these algorithms at runtime so that it is possible to compare the time of reconfiguring a single algorithm *versus* switching between two instances of the same algorithm with different configurations. It was also necessary to provide the BFT-SMaRt with support for the parallel execution of multiple algorithms; this has been achieved by adding new fields to the message headers so that the dispatch layer is able to forward the messages to the right algorithms. Finally, support for including, in the messages, a proof that the reconfiguration was requested was added. This allows the activation by necessity of algorithms in replicas that, due to the asynchrony of the system, have not yet received these commands directly from the replicas of the adaptation manager.

Note that it would be possible to optimize switching by having specific switches for different types of faults (for example, by implementing the protocols described in [7] or [18] to switch between two fault-tolerant configurations). However, the current protocol tolerates both Byzantine and crash faults, and can be used to switch between the various possible configurations.

In developing this extension to BFT-SMaRt, we had the care to integrate them in the best way in the current development trunk. For example, BFT-SMaRt was already prepared to receive some reconfiguration commands, although only for some specific adaptations such as the run-time change of the replica set. We have developed our support mechanisms for adaptation, particularly the switch, as an extension and generalization of these services. The implementation of the new algorithm also followed the class decomposition already used in the implementation of the Mod-SMaRt. A representation of the overall architecture of the system is presented in Figure 1. These options will facilitate the maintenance of our code and its future inclusion in the BFT-SMaRt distribution. The developed code corresponds to approximately 2k new lines of code (the base distribution of the BFT-SMaRt has about 28k lines). A significant part of the development effort of our extensions consisted in the need to understand in depth the source code of the BFT-SMaRt to ensure the correct integration of the new extensions.

Specially, developing an adaptable algorithm from BFT-SMaRt was the most complex task. BFT-SMaRt was already

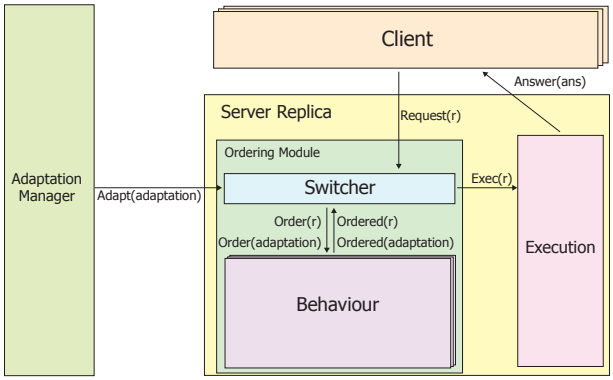


Figure 1: System architecture

optimized for Mod-SMaRt and introducing Fast-SMaRt not only broke performance as it also broke correctness. This happened because some parts of BFT-SMaRt were implemented with the Mod-SMaRt execution flow in mind and didn't work with other consensus behaviour, like the leader election. This, among other sections, had to be reimplemented in order to cope with Fast-SMaRt, and keep performance as close as possible to the original BFT-SMaRt.

V. Evaluation

In order to compare the different adaptation techniques, in this section we intend to answer the following questions:

a) How much time it takes for a reconfiguration to be executed with each technique?
b) What is the impact on system performance caused by adapting with the different techniques?
c) What load is imposed by the adaptation on the network?
d) How useful are the the distinct adaptation techniques to bring the system out of a poor performance situation caused by environmental conditions?
e) How large is the advantage of using reconfigurable protocols, instead of stoppable or black-box, if on wants to perform adaptations that do not demand coordination (known as flash adaptations)?

So, in this chapter an experimental evaluation is presented to answer such questions. In §V-A and §V-B we address questions a), b) and c). Question d) is discussed in §V-D and question e) is explored in §V-C

To mitigate artefacts in the data caused by unwanted hidden variables, like the physical network conditions or the virtual machines performance, 15 runs were performed for each experiment. The 4 most deviating datasets, the two with smaller value and the two with bigger values, were discarded. After that an average was calculated, being the values presented in the graphs. The standard deviation fell between 7% and 12%, nevertheless the differences among the different techniques kept consistent within each of the individual experiment run.

A. Local Network Context

To execute this experiments 6 BFT-SMaRt replicas were used, as this is the minimum replicas needed to tolerate one fault using Fast-SMaRt [17], and a client capable of introducing variable load in the system. All the replicas and the client were hosted in independent virtual machines on the DigitalOcean[1] cloud provider. Each machine as a dual core CPU running at 2.40GHz, 2GB of RAM and a *full-duplex* 1Gb/s network connection. This configuration was chosen for being the most powerful, hence closer to a real-world server, within the available resources. The client which generates load sends requests in multiple threads, simulating multiple clients. In this section of experiments the client sends a request of 10kB each after receiving the reply (with 10B) of the previous request (works in a *closed-loop*). The system ran for 4 minutes, which were dropped, before any adaptation was executed so that the load introduced by the client could put the system in a stable point of performance. The same experiences were also performed with 11 replicas (tolerating two faults) and, although the performance of the system was 62% inferior, the conclusions were similar. The results are omitted due to space restrictions, being available in the thesis.

1) Adaptation Time: To evaluate the time needed to perform an adaptation using each technique, it was accounted the delay between the arrival of an adaptation request and the application of such adaptation. This has been performed with different amounts of load introduced in the system. The results are presented in Figure 2. In the graph is also represented the time of a consensus run to order a request, since it represents the minimum time to execute an operation on the system with the given conditions. It is observable that the time of adaptation using reconfigurable protocols grows much slower that the other approaches, closely to the consensus time. This happens mainly because in a solution using reconfigurable protocols it's possible to have access to the queue of incoming requests and prioritize adaptation requests, while in all other techniques that is not possible. This way, when using reconfigurable protocols, the adaptation time grows only with the time taken to order a request, while on the other hand, in the other techniques it grows both with the time to process a request and the queuing time.

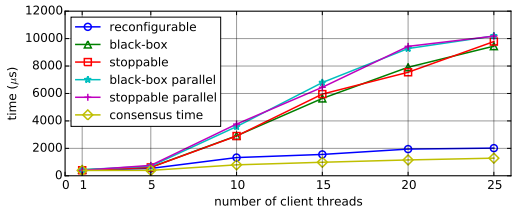


Figure 2: Execution time of an adaptation in a local network, using 6 replicas

[1]https://digitalocean.com

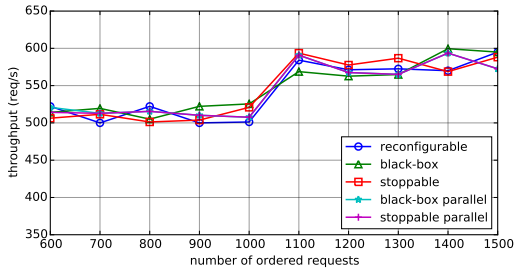


Figure 3: Throughput during an adaptation in a local network with 6 replicas

2) Throughput During an Adaptation: To measure the overhead of executing an adaptation with each of the techniques described, it was used a client with 20 threads, being this the configuration that introduced a significant load in the system without putting it under excessive stress. The throughput was extrapolated at every 100 requests ordered, by measuring the time it took to order the said requests. After the request number 1000 an adaptation request, to switch from Mod-SMaRt to Fast-SMaRt, was submitted to the system. The obtained results are shown in Figure 3.

It can be observed that, in the context of this experiments, none of the used techniques introduced a visible overhead in the throughput of the system. The rise in throughput between requests 1000 and 1100 exists because the adaptation introduced in the system contributed to an overall increase in performance.

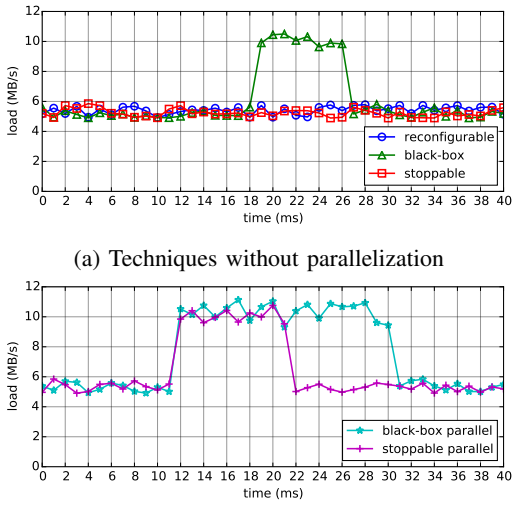
3) Network Overhead of an Adaptation: The network load was measured during the same experiment described above, with samples every millisecond. In order to facilitate the comparison of the data, the most relevant 40ms of execution were taken and aligned so the reconfiguration request arrives at 10ms. The data is represented in Figure 4.

On the other hand, the load introduced in the network varies among the different techniques used. The use of non reconfigurable protocols imposes an increase on network usage after an adaptation is executed as the protocol that ceased to be active continues to execute in the background until it depletes the queue of received requests. When using parallelization techniques there is a visible overhead on the network prior to the execution of a reconfiguration. This happens because as soon as an adaptation request is received the next protocol to be active starts executing tentatively in parallel with the currently active protocol, existing a period with an increase of nearly 90% of the load on the network, when comparing with the normal execution.

B. Wide-area Network Context

To experiment the different techniques in an wide-area network context, a cross-datacenter network was emulated. To emulate the said network topology 6 replicas were hosted in the same datacenter and latency was introduced between them at the Linux Kernel level, using the *netem*[2] tool. To

[2]https://wiki.linuxfoundation.org/networking/netem



(a) Techniques without parallelization

(b) Techniques with parallelization

Figure 4: Network load during an adaptation, in a local network, using 6 replicas

Table I: Latencies between the different replicas in the system used in the wide-area network experiments

Replica	1	2	3	4	5
0	10ms	74ms	84ms	52ms	89ms
1	-	69ms	79ms	45ms	81ms
2	-	-	10ms	107ms	154ms
3	-	-	-	118ms	161ms
4	-	-	-	-	52ms

try to have a network environment as close as possible to a realistic use case, real latency values across Amazon datacenters were used. The emulated network simulates having a system replica in each of the following datacenters: North California, Oregon, Ireland, Frankfurt, Tokyo and Sidney. The amount of latency between each replica is presented in [19] and are shown in Table I. To emulate the variability in the communications delay a jitter of $\pm 10\%$ of the latency added.

1) Adaptation Time: Except from the network environment, this experiment closely followed the one described in §V-A1. However, due to the increase of latency, the batching mechanism of BFT-SMaRt was more prominent, reducing the load of consensus messages to be processed. This raised the need to introduce more clients to introduce enough load to bring the system performance to its peak. So, 2 multi-threaded client were used using a total number of threads between 100 and 1300, with half of the threads running in each client process. The collected data is shown in Figure 5.

The results show that the difference in time between adapting using a reconfigurable protocol *versus* adapting with other techniques is much closer to be constant than in a local network, mainly up to the load introduced by 1100 replicas. This happens because the latency introduced by the network (felt by all the techniques) is much more

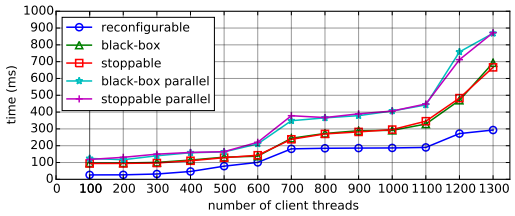


Figure 5: Reconfiguration time in a wide-area network environment

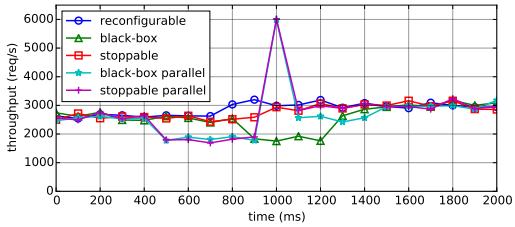


Figure 6: Throughput during an adaptation, in an wide-area network, using 1000 client threads.

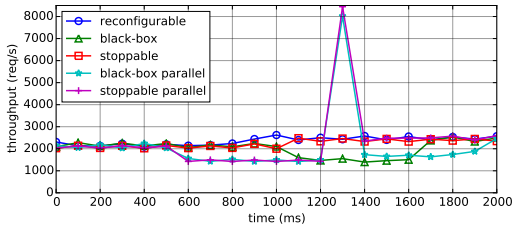


Figure 7: Throughput during an adaptation, in an wide-area network, using 1200 client threads.

relevant than the latency introduced by the queuing time, which is the main differentiator of the different techniques reconfiguration times. Nevertheless, the absolute value of the difference between the distinct solution is much more prominent than in a local network, instead of less than ten milliseconds, in this environment the differences ar hundreds of milliseconds.

The steeper slope observed starting in the load introduced by 1200 clients derives from the fact that the system starts to fail in coping with the number of incoming requests, having greater queuing times, which affects directly the time of executing an adaptation when using non reconfigurable or stoppable protocols.

We can then conclude that in a wide-area network environment, there is a greater load span in which the different techniques present very significant differences, when compared to a local network. However differences exist in the overall time, and may get several orders of magnitude different if the queuing time surpasses the ordering time of a request. This can be of interest if the adaptation is time-sensitive and must be performed as fast as possible, as in a local network, using a reconfigurable protocol is the faster technique.

2) Throughput During an Adaptation: To test the overhead of carrying an adaptation in an wide-area network environment a similar method to the one described in V-A2 was followed. 6 BFT-SMaRt replicas and 2 multi-threaded clients were used. Because of the higher latency in request processing it was possible to collect data with time based samples. This is, instead of collecting data samples every 100 requests ordered, like in the local network experimental method, the throughput was calculated at every 100ms based on the number of requests answered in that time. This approach facilitates the interpretation of the results as it's more intuitive to rationalize using the passage of time instead of the number of requests answered. After 4 minutes, 20 seconds of execution were registered, where an adaptation request arrived at the system shortly after the 500th millisecond. The throughput of answered requests during this time, with a load of 1000 and 1200 client threads, is represented in Figures 6 and 7. This loads were chosen because 1000 clients introduced a steady peak performance and 1200 clients introduced a near-peak performance with an increase in queuing time.

As the adaptation takes more time in this network environment the differences in performance among the different techniques become more apparent. One can note that the adaptation using reconfigurable and stoppable protocols carry little to no overhead in throughput. On the other hand, solutions using non-reconfigurable protocols carry a penalization in throughput in the moments after an adaptation, because the protocol that ceased to be active continues to operate after the adaptation, until the it depletes the existing queue of incoming requests, hence wasting processing resources. Finally the approaches that use parallelization present a belly before the executing of the adaptation and a peak in throughput right after the adaptation is executed. The loss of throughput happens because of the parallel execution of protocols in these techniques, carrying costs in performance due to the sharing of resources among both protocols. The peak happens because the new protocol already started ordering requests as soon as the adaptation command arrived and now it can dispatch in burst all the processed requests to the clients.

The behaviour of parallelization techniques arise the question if the peak reached after the adaptation compensates the loss of performance before it, when compared to their counterpart techniques that use no parallelization. In order to answer this question a graph showing the total amount of requests answered with the different techniques was derived from the throughput data. The results are presented in Figures 8 and 9.

The data shows us that in the case of using black-box protocols the parallelization optimization reveals itself to be of worth, compensating, in part, the penalization introduced by the use of black-box protocols when compared to reconfigurable and stoppable ones. Although the difference is only about 100 requests in this experiment, this small

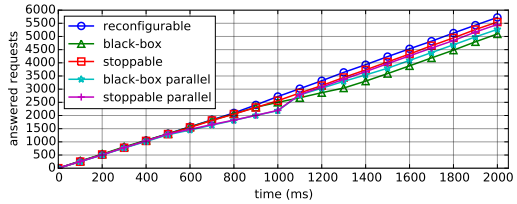


Figure 8: Answered requests during an adaptation, in an wide-area network, using 1000 client threads.

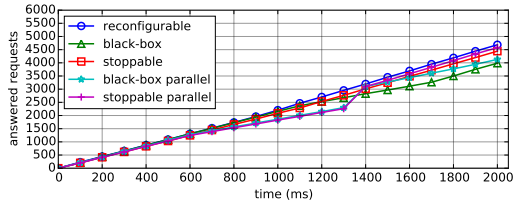
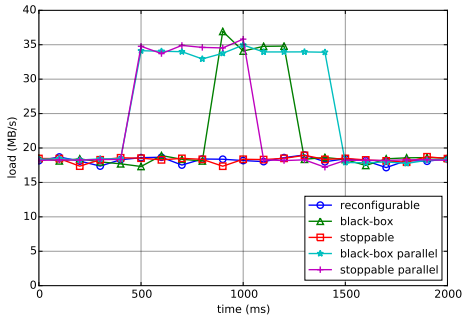


Figure 9: Answered requests during an adaptation, in an wide-area network, using 1200 client threads.

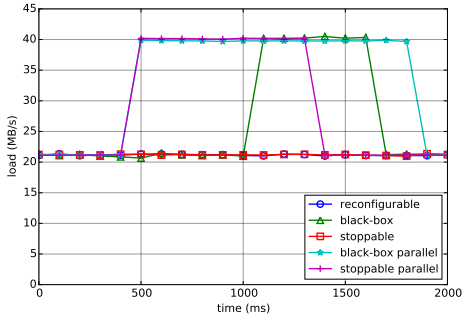
differences may add up and impact a long-lived system with recurring adaptations. On the contrary, when comparing the use of parallelization when using stoppable protocols the data shows that not always the parallel execution is beneficial. This happens because using stoppable protocols does not incur in visible losses of performance by itself. So, the issue relies on if the peak of throughput using stoppable protocols with parallelization compensates the loss of performance prior to the adaptation because there is no loss of performance to cover, like when using black-box protocols.

We can see that parallelization surpasses its counterpart when using 1200 client threads, while it doesn't when using 1000 client threads. The main difference among them is the time between receiving an adaptation request and executing it, which is the time that the soon-to-be active protocol executes tentatively. It's visible that with a greater time, the parallelization technique has better results. This happens because the consensus protocols usually have a warm-up time until they reach peak performance, in the Fast-Smart case its due to the batching behaviour, which benefits with queues which length is near the maximum batch size. So, with shorter adaptation times, the time spent warming up the new protocol may not compensate the overall loss of performance due to parallelization. In the conditions of this experiment, a possible optimization for adapting using stoppable protocols and parallelization would be to delay the execution of the adaptation so the protocol executes enough time to compensate the overall drop in throughput, contrary to the intuitive idea of executing an adaptation as fast as possible. Of course this is a very specific case, where the adaptation is being made only to increase performance, not being critical or time sensitive as it would be if it was done for security purposes, for example. Furthermore, the peak throughput of the protocol executing tentatively must be higher than loss of throughput in the active protocol when compared to the throughput it would have if no other protocol was executing.

3) Throughput Overhead of an Adaptation: During this experiments described above, the network load was also collected. The results are shown in Figure 10. Although the differences are amplified due to greater load and adaptation time, the conclusions are similar to the ones discussed in V-A2.



(a) With a load of 1000 clients.



(b) With a load of 1200 clients.

Figure 10: Load introduced in an wide-area network when executing an adaptation

C. Adaptation Under Heavy Load

All the previous discussed experimental cases assumed a stable execution of the system, this is, the system was not operating in conditions that aggressively reduced its performance. This leads to the question: *How useful are the the distinct adaptation techniques to bring the system out of a poor performance situation caused by environmental conditions?*

To answer this question, a simulation of several clients becoming faulty, and introducing too much load, was done to test how the adaptations could help dealing with it. Specifically, the experiment simulates clients who operate out of the BFT-SMaRt protocol by executing in an open-loop, this is, not waiting for the answer to request n before sending request $n + 1$. To emulate a possible adaptation issued by the adaptation manager, a protocol switch to Safe-SMaRt was issued when the system's latency degraded

beyond 43 seconds. This happened at second 0 in the graph presented in Figure 11.

The network conditions are similar to the ones used in V-B. There were used 1000 clients were half of them, 500, were faulty and operating in an open-loop, ignoring the answers of the replicated server.

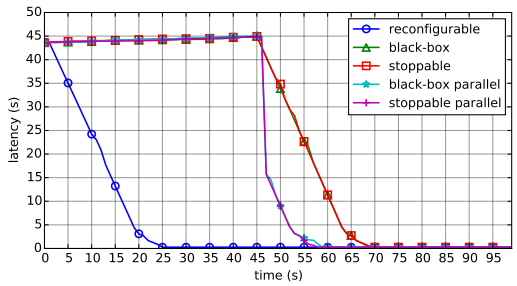


Figure 11: Introduction of an adaptation when the system's performance is heavily degraded

Note that this experiment does not aim at proving the utility of Safe-SMaRt improve the resilience of the system to faulty clients, as that falls out the scope of this thesis. This experiment was performed to compare how each adaptation technique performs when used in heavily degraded system's performance conditions.

It is possible to note that using adaptable protocols is again the fastest way to introduce an adaptation, being introduced more than 40 seconds before any other technique in this case. This happens because of the prioritization of adaptation requests, as discussed before. A new insight, not verified clearly before, is that in this case, the parallelization performs better that non-parallel commutation. Even if they are applied some seconds later, their parallel execution helps to bring the system to a good performance much faster. This happens because the new protocol, executing in parallel, has the chance of operating during a considerable amount of time, being able to operate in peak performance for a great amount of time. Moreover, the new protocol is more resilient to the attack being performed, having better performance than the previous protocol. Note that although this is verifiable in this case, in some performance degradation contexts, parallelization may be worse than non-parallel techniques, specially if the bottleneck in performance is bandwidth or computing power.

D. Flash Adaptation versus Ordered Adaptation

As seen in III-B, when using reconfigurable protocols it is possible to introduce fine-grained adaptations. Even more so, some of this adaptations can be performed without the necessity of running a consensus. This arises the question *How great is the advantage of using reconfigurable protocols, instead of stoppable or black-box, if on wants to perform adaptations that don't demand coordination?* To answer this, the delay of introducing an adaptation which does not require a consensus (called flash adaptation) using a reconfigurable protocol *versus* using other techniques, namely stoppable and pure black-box protocols. The adaptation introduced aimed at start logging the replicas activity to disk. A fine-grained adaptation which demands coordination, specifically changing the leader, is also present in the chart to allow a comparison between a coordination-demanding and a flash adaptation using a reconfigurable protocol.

The experimental setup follows closely the one used for wide-area performance experiments (V-B), but with added latency between the adaptation manager's replicas and the system replicas. This latency was added using netem, as before, with values between 25ms and 35ms. The time was measured between issuing the first adaptation command at the adaptation manager's replicas and it being applied by a quorum of system's replicas, which consists in 5 replicas in this context. The last replica was ignored as it could be faulty, under the system model. The obtained results are presented in Figure 12.

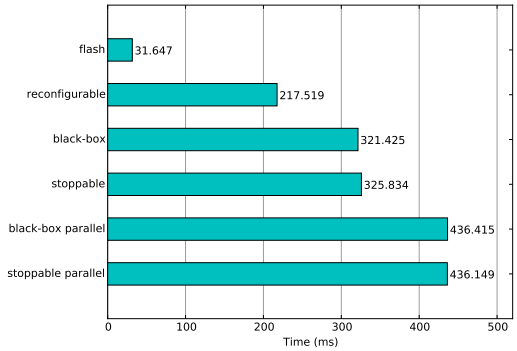


Figure 12: Adaptation time of flash, coordination-demanding, stoppable and pure black-box approaches.

It is visible that flash, using reconfigurable protocols, adaptations are applied much faster than when the same adaptation is applied with other techniques. This follows what was expected, as when using stoppable and black-box protocols this adaptations still demand running consensus to switch between two protocols with different configurations, whereas when using reconfigurable protocols it does not.

The majority of the time taken by the flash technique to apply the configuration is actually the latency between the adaptation manager's replicas and the switchers, as applying the adaptation is in practise just some simple method calls. On the other hand, when applying a coordination-demanding adaptation, it takes more time due to the need of a consensus run. The differences among the other techniques derive from the loss of performance when using parallelization and the queueing time, as already previously discussed in V-B.

VI. Conclusion

In this dissertation we have addressed the problem of performing dynamic reconfiguration of BFT systems. We have organized, in a systematic manner, a portfolio of algorithms to switch in run-time between different protocols. The algorithms have been derived by adapting previous solutions that have been developed for different fault models.

The algorithms can be applied in different scenarios, depending of the properties of the protocols involved in the reconfiguration. We have classified the target protocols into three main categories, namely: reconfigurable protocols, stoppable protocols, and protocols without any specific support for adaptation, that need to be treated as black-boxes. We have also identified several optimizations that can be applied to the reconfiguration algorithms, such as prioritizing adaptations and using parallelization.

To understand the tradeoffs involved when applying these algorithms in practice, and to obtain a comparative assessment of their performance, we have implemented them in a common framework, based on the BFT-SMaRt open source project. We have deployed these implementation in different settings.

Our results show that in a local network, using target protocols as a black-box reveals itself to be the best solution given that it presents the same performance as other alternatives and it is the easiest to deploy. On the contrary, when performing reconfiguration in a geo-replicated system, the use of black-box protocols presents a significant overhead when compared to other alternatives. Thus, it may be worth to change to target protocols such that they support at least a stoppable interface. We have also observed that parallelization is always useful for high-bandwidth networks when using black-box algorithms. However, if stoppable algorithms are used, parallelization is only beneficial in face of large network delays. Finally, as as expected, if the target protocols are reconfigurable, switching can be executed with significant savings, namely in terms of latency.

As future work we would like to use the findings reported here to improve adaptation policies for several concrete applications that can benefit from dynamic reconfiguration, such as geo-replicated distributed ledgers. Another interesting direction in future work is to combine different switching algorithms in order to switch between protocols with distinct levels of support for adaptation.

ACKNOWLEDGMENTS

This research has been supported in part by FCT through projects 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, Manuel Bravo and Daniel Porto. Some parts were also performed in collaboration with members of Large-Scale Informatics Systems Laboratory of Faculdade de Ciências da Universidade de Lisboa, namely, Alysson Bessani and João Sousa.

REFERENCES

[1] F. Schneider, "Replication management using the state-machine approach," in *Distributed Systems, 2nd Edition*, ser. ACM-Press, S. Mullender, Ed. Addison-Wesley, 1993, ch. 7.

[2] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[3] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2009.

[4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of 21st Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

[5] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire," in *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2008.

[6] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," *IEEE Transactions Parallel Distributed Systems*, 2015.

[7] J. Mocito and L. Rodrigues, "Run-time switching between total order algorithms," in *Proceedings of the Euro-Par 2006*, ser. LNCS. Springer-Verlag, 2006.

[8] W. Chen, M. Hiltunen, and R. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2001.

[9] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014.

[10] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, 2010.

[11] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman, "Reconfigurable state machine replication from non-reconfigurable building blocks," *arXiv preprint arXiv:1512.08943*, 2015.

[12] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable paxos," Microsoft Research, Tech. Rep., 2008.

[13] P. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Transactions on Computer Systems (TOCS)*, 2015.

[14] M. Pasadinhas, D. Porto, A. Lopes, and L. Rodrigues, "Adaptação guiada por políticas de sistemas tolerantes a faltas bizantinas," in *Actas do 9º Simpósio de Informática (Inforum)*, Aveiro, Portugal, Oct. 2017.

[15] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., 2006.

[16] J. Sousa and A. Bessani, "From byzantine consensus to bft state machine replication: A latency-optimal transformation," in *Ninth European Dependable Computing Conference (EDCC)*. IEEE, 2012.

[17] J. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions Dependable and Secure Computing*, 2006.

[18] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues, "Visigoth fault tolerance," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*. ACM, 2015.

[19] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*. ACM, 2017.