Dynamic Adaptation of Byzantine Consensus Protocols

Carlos Carvalho[1], Daniel Porto[1], Luís Rodrigues[1], Manuel Bravo[1,3], Alysson Bessani[2]
[1]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[2]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[3]Université Catholique de Louvain, Belgium

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 or changed at runtime 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 library. Using this 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.

CCS CONCEPTS

• **Computer systems organization → Self-organizing autonomic computing**; **Dependable and fault-tolerant systems and networks**;

KEYWORDS

Byzantine Fault Tolerance, Consensus, Dynamic Adaptation

ACM Reference format:
Carlos Carvalho[1], Daniel Porto[1], Luís Rodrigues[1], Manuel Bravo[1,3], Alysson Bessani[2]. 2018. Dynamic Adaptation of Byzantine Consensus Protocols. In *Proceedings of ACM SAC Conference, Pau,France, April 9-13, 2018 (SAC'18),* 8 pages.
https://doi.org/10.1145/3167132.3167179

1 INTRODUCTION

State Machine Replication (SMR) [20] 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 [5], Aardvark [7], and Zyzzyva [12]. Each of these systems operates better under certain conditions, and worse in others, with none surpassing all others in all situations, as shown in [21]. 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 the 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 [21], 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, 8, 16]). 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 [2]. 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.

SAC'18, April 9-13, 2018, Pau,France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167179

2 RELATED WORK

Dynamic reconfiguration have been widely studied for systems that require adaptability and/or high availability [1, 3, 6, 8, 9, 11, 13, 14, 16, 19, 23]. In this section, we focus on the multiple techniques proposed for dynamically adapting consensus protocols. We group these adaptive protocols into two categories: those using a single *reconfigurable consensus protocol*, those using *multiple 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. For instance, Santos de Sá et al. propose a reconfigurable variant of PBFT [5] that allows for online changes in the batching size and timeouts, in order to adapt, for instance, to changes in the workload characterization [9]. Since consensus protocols involve multiple processes, it is sometimes 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); e.g., when changing the communication pattern. Lamport et. al. described how this technique can be carried out safely [14].

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

The advantage of this type of adaptation is that any protocol could be integrated without major effort, as it 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. This approach has been studied by previous work [3, 16]. Mocito et al. [16] assume a perfect failure detector; while Bortnikov et al. [3] presented a variant that requires no perfect failure detector.

Switching can be facilitated if the protocols export an interface that allows the switcher to deactivate the protocol, placing it in a quiescent state [14]; in the literature, these protocols are designated as stoppable [13]. In this case, after the deactivation has been requested by the switcher, the 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 replicas. Furthermore, not all available Byzantine consensus protocols support deactivation, which limits the coverage of this approach.

Aublin et. al. [1] presented a special type of stoppable protocols. 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 system development framework that allows to deactivate one algorithm and replace it with another one when the 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. [16] and Bortnikov et. al. [3] 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 as all messages are ordered by the two protocols. Thus, this optimization can only be applied in systems in which the network does not present a bottleneck.

3 BYZANTINE PROTOCOL ADAPTATION

In this section, we discuss the operation of several protocol switching strategies in the presence of Byzantine faults. We classified these strategies in two categories: i) *adaptation using a reconfigurable algorithm*, and ii) *adaptation using algorithms as black-boxes*, matching the categories described in §2. 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.

3.1 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 asynchronous periods may exist, but there is always unknown synchronous periods in which the system can make progress.

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 a command to all the switchers to start the adaptation. These 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 [17]; 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. There are many reasons to trigger a reconfiguration: adapt to changes in the workload, change the leader replica of a protocol because of performance or security issues, changing the consensus behaviour to optimize it to current conditions like network stability and failure frequency, and many more.

3.2 Using Reconfigurable Algorithms

As seen in §2, when using a reconfigurable algorithm, the main challenge is to coordinate all the replicas to use the same configuration. One of the simplest ways of doing it, as suggested in [14], 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 correctness of the switching algorithm depends exclusively on the properties of the underlying consensus algorithm. Therefore, leveraging BFT consensus to reconfigure a Byzantine fault tolerant protocol, leads to BFT reconfigurations. This technique is potentially the most efficient, especially since the protocol can be optimized to prioritize reconfiguration requests, making these messages to be ordered before other regular application messages enqueued for ordering.

Unfortunately, efficiently implementing reconfigurable consensus algorithms is hard. One of the problems, apart from the obvious complexity of such 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 §4.

3.3 Using Algorithms as a Black-Box

We divide this category into two subcategories, depending whether the adaptation mechanism relies on *stoppable* or *non-stoppable* consensus protocols. Finally, we describe how a possible optimization to both subcategories that aims at reducing the impact of reconfiguration by running multiple algorithms simultaneously.

Using non-stoppable algorithms. 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), the switcher starts 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, when using non-stoppable algorithms, 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 challenge to non-stoppable algorithms, in consequence of the presence of Byzantine faults, is to verify that the marker corresponds to a command that was actually issued by the adaptation manager. The goal is to prevent a Byzantine switcher from inducing undesirable reconfigurations. We have considered two techniques for achieving this. 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 for the proof (since by waiting for $f+1$ markers, we are sure that at least 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.

Using stoppable algorithms. Adaptive strategies that use stoppable consensus algorithms have the potential of simplifying the reconfiguration protocol at the switchers. When a deactivation is requested, these algorithms ensure that no request is ordered after the stop command. Therefore, the switcher does not have to implement — as for the non-stoppable algorithms — 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 ordered by the adaptation 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 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 support deactivation, which limits the coverage of this approach.

In the Byzantine case, there is an additional 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 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-election processes in protocol B even before its operation is initiated, which can harm system's performance. For this reason, in a BFT system, it may be preferable for a switcher replica to immediately send messages in protocol B, even before making sure that 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.

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 (e.g., [3, 16]), 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 via protocol B for the first time only activates this protocol 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 practice the system will perform better when using this technique as it is resource-intensive and can lead to starvation.

3.4 Discussion on the Techniques

Reconfigurable algorithms have the potential of exhibiting the quickest transition among configurations. Nevertheless, this technique requires that all desirable behaviours are supported by a single protocol, which quickly becomes extremely complex to maintain, and whose correction is difficult to ensure. On the other hand, using multiple algorithms as black boxes is a more modular solution that a priori is easier to extend and requires less integration effort. Among them, solutions requiring non-stoppable capabilities are the easiest to maintain and extend, as any consensus algorithm could be used. Nevertheless, more coordination among replicas is required and therefore poorer performance is expected.

Our experience implementing these techniques match these insights regarding the maintenance and integration effort. Our experimental evaluation does a more throughout and precise comparison of the performance exhibited by these techniques under different settings.

4 BFT-SMART INSTANTIATION

In order to carry out an experimental evaluation, required to support a comparative analysis of the performance in practice of the various techniques, we have implemented the different algorithms in a common software framework, namely, the BFT-SMaRt library [2]. 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 few complete and actively supported BFT SMR implementations. BFT-SMaRt integrates a single consensus algorithm, a variant of the algorithm described in [10] for the Mod-SMaRt framework [22].

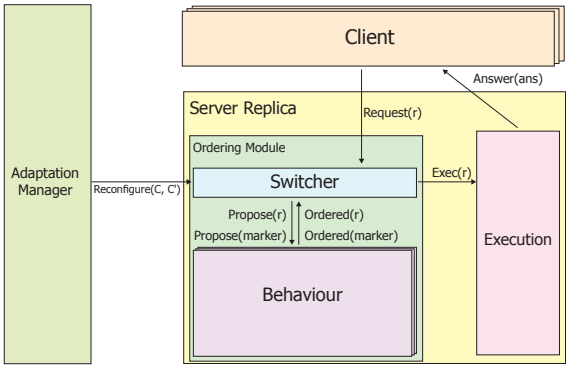


Figure 1: System architecture.

Extensions. We have implemented a set of extensions to BFT-SMaRt that are relevant for our experiments:

- We have integrated a switcher that mediates between clients and the supported consensus algorithms. Our switcher is oblivious to the causes triggering the reconfiguration. Nevertheless, one could implement specialized switchers that minimize the switching time at the cost of having to implement one for each type (or group of) reconfiguration commands. For instance, by implementing the protocols described in [16] or [18] to switch between two fault-tolerant configurations.
- We have integrated an implementation of the Fast Byzantine Consensus [15], which we call Fast-SMaRt. The goal is to have more than one algorithm to switch among.
- We have developed stoppable versions of Mod-SMaRt and Fast-SMaRt. Thus, we can measure the cost of switching between algorithms with support for deactivation.
- We have made both algorithms reconfigurable. Thus, one can reconfigure the leader of these algorithms at runtime. The goal is to be able to compare the time of reconfiguring a single algorithm *vs.* 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 a new field (1B) to the message header identifying the algorithm to which the message is addressed such that the dispatch layer is able to forward the messages to the right algorithm.
- Finally, we also add support to include in adaptation messages a proof that the reconfiguration was indeed requested and not created by a malicious party. Thus, when a switcher has received enough requests (a quorum of identical requests) from the adaptation manager replicas, it generates a veracity proof and submits the reconfiguration command (together with the proof) to the active consensus protocol. Other switchers can then trust that the reconfiguration command was indeed requested by at least one correct adaptation manager replica. This speeds up the reconfiguration by forcing activation at replicas that, due to the asynchrony of the system, have not yet received these commands directly from the replicas of the adaptation manager.

Implementation. When developing these extensions, 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 25k 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 its correct integration. Specially, developing an adaptable algorithm from BFT-SMaRt was the most complex task. BFT-SMaRt was already 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 specifically for Mod-SMaRt execution flow and did not consider other protocols. For instance, we had to heavily modify the leader election to support Fast-SMaRt.

5 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) how adapting affects throughput on each of the techniques? c) what load is imposed by the adaptation on the network?

To execute these experiments six BFT-SMaRt replicas were used, as this is the minimum number of replicas needed to tolerate one fault in Fast-SMaRt [15][1], and clients capable of introducing variable load in the system. All the replicas and the client were hosted in independent virtual machines on the DigitalOcean cloud provider.[2] Each machine has a dual core CPU running at 2.40GHz, 2GB of RAM and a *full-duplex* 1Gb/s network connection. The load generator process sends requests through a number of 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 (this is with zero thinking time, i.e., in a *closed-loop*). Both of the consensus protocols used throughout the evaluation (Fast-SMaRt and Mod-SMaRt) have been configured with a 1024 maximum batch size, retransmission timeout of one second and a leader accusation timeout of 2 seconds. Neither protocol uses a batching timeout as the leader builds a new batch every time a previous consensus run finishes. Each experiment ran for 5 minutes before any adaptation was executed such that the load introduced by the client could put the system in a stable point of performance. The first minute was also dropped.

5.1 Local-Area Network Context

Firstly we tested the distinct solution in a local-area network context, simulating the case where all servers are hosted within the same datacenter. The observed round-trip time of messages sent from replica to replica never exceeded half millisecond. Only one load generator was used to simulate between 1 and 25 clients.

[1]The experiments were also performed with 11 replicas (tolerating two faults) and we verified that, although the throughput was on average 62% inferior, the overall pattern is very similar, the results are omitted due to space constraints, being available in [4].
[2]https://digitalocean.com

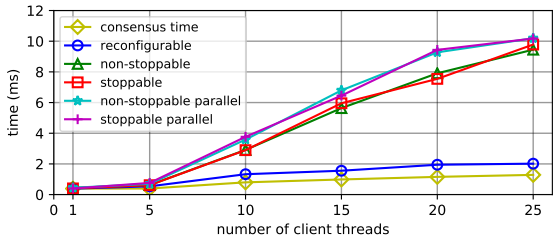


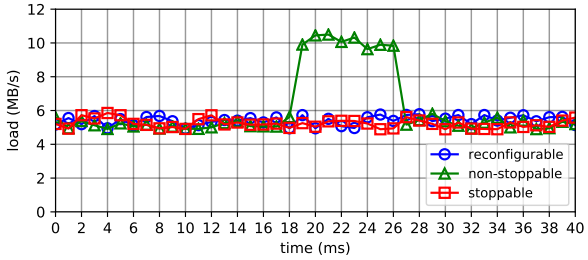
Figure 2: Execution time of an adaptation in a LAN.

Figure 3: Throughput during an adaptation in a LAN.

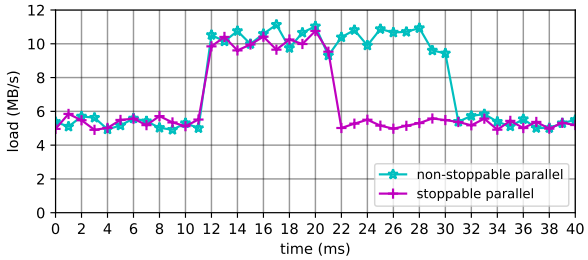
5.1.1 Adaptation Time. To evaluate the time needed to perform an adaptation using each technique, we measure the time between the instant in which the switcher receives a quorum ($f + 1$) of identical reconfiguration requests from the adaptation manager replicas and the execution of such reconfiguration. We vary the number of clients to observe the effect that different loads have in the adaptation time. The results are presented in Figure 2. The time of applying a reconfiguration is the sum of the time the request spends in queue, the consensus run time and its execution. We verified, in our experiments, that the execution time is negligible. As a baseline, the time of a consensus run was added to the graph. This represents the best case for processing a request, when all request queues are empty.

It is observable that the time of adaptation using reconfigurable algorithms grows much slower than the other approaches. This happens mainly because reconfigurable algorithms can have access to the queue of incoming requests and prioritize adaptation commands, while in all other techniques this is not possible. This way, when using reconfigurable algorithms, the adaptation time grows only with the time taken to order a request, while in the other techniques, it grows both with the time to process a request and the queuing time.

5.1.2 The Effect of Adaptation on Throughput. To measure how adapting affects the throughput exhibited by the different techniques, we run a set of experiments in which the system switches from the Mod-SMaRt algorithm to the Fast-SMaRt after 1000 client requests. We set the number of client to 20, 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 these requests. The obtained results are shown in Figure 3.



(a) Techniques without parallelization



(b) Techniques with parallelization

Figure 4: Network load during an adaptation in a LAN.

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

5.1.3 Adaptation Network Overhead. 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 such that reconfiguration request arrives at the 10^{th} millisecond. The results are shown in Figure 4.

Unlike the previous experiment, the load introduced in the network varies among the different techniques. The use of non-stoppable algorithms imposes an increase on network usage after an adaptation is executed as the algorithm that ceased to be active continues to execute in background until it depletes the queue of received requests. This extra load is perceived after the reconfiguration was ordered and applied, around the 18^{th} millisecond for no parallelization and the 22^{nd} when using parallelization.

Also, when using parallelization techniques as an optimization for the black box techniques (stoppable and non-stoppable) there is a visible overhead on the network prior to the execution of a reconfiguration. This occurs because as soon as an adaptation request is received, the next algorithm to be active starts executing tentatively in parallel with the currently active one, leading to a period with an increase of nearly 90% on the network load, when comparing with the normal execution.

5.2 Wide-Area Network Context

To experiment the different techniques in an wide-area network context, a cross-datacenter network was emulated using 6 replicas hosted in the same datacenter and introducing latency at the

Table 1: Latencies among replicas

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

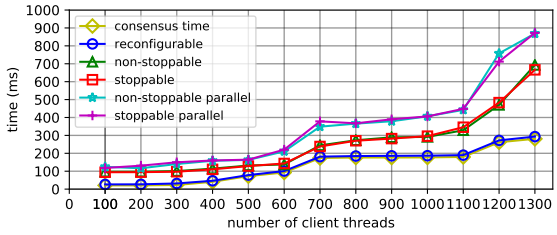


Figure 5: Reconfiguration time in a WAN.

Linux Kernel level, using the *netem* tool.[3] To emulate a realistic setting, real latency values across Amazon regions were used. The experiments emulate a network that has a system replica in each of the following datacenters: North California (0), Oregon (1), Ireland (2), Frankfurt (3), Tokyo (4) and Sidney (5). We measured empirically the latency between Amazon regions and the latency between each pair of replicas is shown in Table 1. To emulate the variability in the communications delay, a jitter of ±10% of the latency was added. Due to the increased latency in the system, the batching mechanism of BFT-SMaRt was more prominent, processing more requests per consensus run. This raised the need to use two load generator processes to make the system reach its peak performance, by simulating between 100 and 1300 clients.

5.2.1 Adaptation Time. Except from the network environment, this experiment closely follows the one described in §5.1.1. The collected data is shown in Figure 5.

The results show that the difference in time between adapting using a reconfigurable algorithm *versus* adapting with other techniques is much closer to be constant than in a local network, up to the load introduced by 1100 clients. This happens because the latency introduced by the network (felt by all the techniques) is much more relevant than the latency introduced by the queuing time, which is the main differentiator of the distinct techniques reconfiguration times. 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 black-box techniques.

We can then conclude that in a wide-area network environment, the differences are greater in terms of absolute value, but grow more similarly for some load span (until the queue time surpasses the ordering time). As in a local network, using a reconfigurable algorithm is the fastest technique.

5.2.2 The Effect of Adaptation on Throughput. In order to measure the impact that reconfiguration has on throughput in a wide-area network environment, we run an experiment similar to the one

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

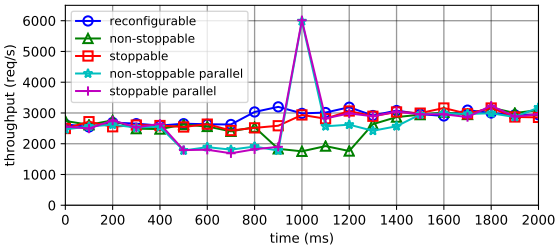


Figure 6: Throughput during adaptation in a WAN, using 1000 client threads.

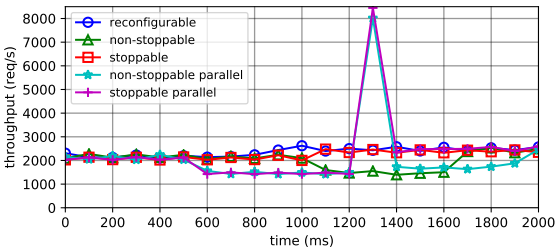


Figure 7: Throughput during adaptation in a WAN, using 1200 client threads.

described in §5.1.2. 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 computed every 100 milliseconds, based on the number of requests answered in that interval. After 4 minutes, 20 seconds of execution were registered, where an adaptation request arrived at the system shortly after the 500^{th} millisecond. The throughput of answered requests during this time, with 1000 and 1200 clients, is represented in Figures 6 and 7. These 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 observe that the adaptation using reconfigurable or stoppable algorithms incurs no significant throughput penalty. On the other hand, solutions using non-stoppable algorithms carry a penalization in throughput right after the adaptation, because the algorithm that ceased to be active continues to operate after the adaptation, until it depletes the existing queue of incoming requests, consuming resources. Finally, the approaches that use parallelization present a depression before the execution of the adaptation and a peak in throughput right after the adaptation is concluded. The loss of throughput occurs in these techniques because two algorithms have to share resources, reducing the overall computational power available to each of them. On the other hand, the peak happens because the new algorithm already started ordering requests as soon as the adaptation command arrived and now it can dispatch all the received requests in a burst.

The behaviour of parallelization techniques rises the following question: does the peak reached after the adaptation compensates

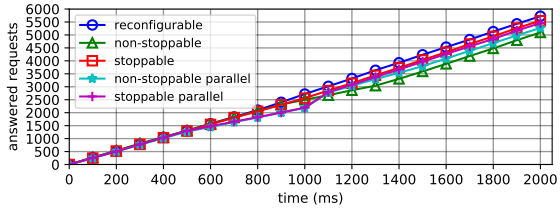


Figure 8: Processed requests during an adaptation in a WAN, using 1000 client threads.

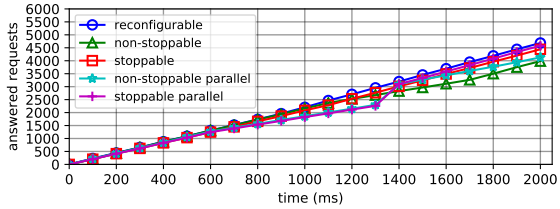


Figure 9: Processed requests during an adaptation in a WAN, using 1200 client threads.

the loss of performance before it, when compared to their counterpart techniques that use no parallelization? In order to answer this question we plot the aggregate of requests processed over time. The results are shown in Figures 8 and 9.

The data shows that when using non-stoppable algorithms, the parallelization reveals itself to be of worth, compensating in part the performance penalty introduced when compared to reconfigurable and stoppable algorithms. Although this difference (of more than 100 requests) is small, in a long-living system, where many adaptations will occur over time, this can have a significant impact.

On the contrary, when using parallelization with stoppable algorithms, the results show that the parallel execution is not always beneficial. We can see that parallelization surpasses its counterpart when using 1200 client threads (134 more requests), while it does not when using 1000 client threads (108 less requests). This happens because, unlike when using non-stoppable algorithms, using stoppable algorithms does not introduce a significant penalty in throughput. The throughput difference of these approaches is caused by the delay between receiving an adaptation request and executing it, which is the time that the soon-to-be active algorithm executes tentatively. It is visible that with a greater time, the parallelization technique has better results. This happens because consensus algorithms usually have a warm-up time until they reach peak performance; in the Fast-SMaRt case its due to the batching behaviour, which benefits from queues length being closer to the maximum batch size. Therefore, with shorter adaptation times, the time spent warming up the new algorithm may not compensate the overall loss of performance due to parallelization. In the conditions of this experiment, a possible optimization for adapting using stoppable algorithms and parallelization would be to delay the execution of the adaptation such that the algorithm executes for enough time to compensate the overall drop in throughput. This conflicts with the intuitive idea of executing an adaptation as fast as possible. Of course, this is a scenario in which the adaptation is triggered only to increase performance, not being critical or time

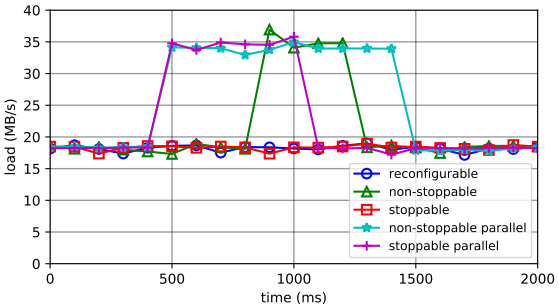


Figure 10: Load introduced in the network (WAN) when executing an adaptation with a load of 1000 clients.

sensitive, as when done for security purposes. Furthermore, the peak in throughput of the algorithm executing tentatively must be higher than the loss of throughput in the active algorithm.

5.2.3 Adaptation Network Overhead. During the experiments described above, the network load was also collected. The results for 1000 client threads are shown in Figure 10 (the results for 1200 threads are omitted due to space constraints). Although the differences are amplified due to greater load and adaptation time, the conclusions are similar to the ones discussed in §5.1.2.

6 CONCLUSIONS

We presented and compared several adaptation techniques for Byzantine fault-tolerant systems. The experimental evaluation performed in a datacenter with a gigabit network shows that none of these techniques significantly penalizes the performance of the system during the adaptation. In this setting, using non-stoppable algorithms is the best approach, as it is the simpler to implement (all consensus algorithms can be used) and carries no loss of performance.

On the other hand, under multi-datacenter deployments, the differences in performance became more apparent. Using non-stoppable algorithms introduces a penalization in throughput and network usage when compared with other techniques. Here, the better trade-off between complexity and performance is using a stoppable approach, which presents no overhead and is relatively simple to implement.

Also, parallelization is only interesting if one is forced to use non-stoppable algorithms and the bandwidth is not a scarce resource. If the network is the bottleneck, parallelizing is not advised, as it consumes extra bandwidth.

Lastly, if the time required to execute an adaptation is key, developing an adaptable algorithm surpasses all other techniques, as it can prioritize the adaptation requests.

For future work we plan to design a framework that is capable to drive adaptation using multiple techniques, mixing reconfigurable, stoppable and non-stoppable algorithms.

ACKNOWLEDGMENTS

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via projects PTDC/EEI-SCR/1741/2014 (Abyss), UID/CEC/50021/2013 and UID/CEC/00408/2013 (LaSIGE).

REFERENCES

[1] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4, Article 12 (Jan. 2015), 45 pages.

[2] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. Atlanta, Georgia, USA, 355–362.

[3] Vita Bortnikov, Gregory V. Chockler, Dmitri Perelman, Alexey Roytman, Shlomit Shachor, and Ilya Shnayderman. 2015. Reconfigurable State Machine Replication from Non-Reconfigurable Building Blocks. *Arxiv preprint arXiv:1512.08943* (2015).

[4] C. Carvalho. 2017. *Dynamic Adaptation of Byzantine Fault Tolerant Protocols*. Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[5] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. New Orleans, Louisiana, USA, 173–186.

[6] Wen-Ke Chen, M. A. Hiltunen, and R. D. Schlichting. 2001. Constructing Adaptive Software in Distributed Systems. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS '01)*. Phoenix, Arizona, USA, 635–643.

[7] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. Boston, Massachusetts, 153–168.

[8] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. 2013. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *Proceedings of the 2013 43th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. Budapest, Hungary, 1–12.

[9] Alírio Santos de Sá, Allan Edgard Silva Freitas, and Raimundo José de Araújo Macêdo. 2013. Adaptive Request Batching for Byzantine Replication. *SIGOPS Oper. Syst. Rev.* 47, 1 (Jan. 2013), 35–42.

[10] Rachid Guerraoui and Luís Rodrigues. 2006. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc.

[11] Jamie Hillman and Ian Warren. 2004. An Open Framework for Dynamic Reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. Edinburgh, Scotland, UK, 594–603.

[12] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. Stevenson, Washington, USA, 45–58.

[13] L. Lamport, D. Malkhi, and L. Zhou. 2008. *Stoppable paxos*. Technical Report. Microsoft Research.

[14] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* 41, 1 (March 2010), 63–73.

[15] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Trans. Dependable Secur. Comput.* 3, 3 (July 2006), 202–215.

[16] José Mocito and Luís Rodrigues. 2006. Run-time Switching Between Total Order Algorithms. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)*. Dresden, Germany, 582–591.

[17] M. Pasadinhas. 2017. *Policy-Based Adaptation of Byzantine Fault Tolerant Systems*. Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[18] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth Fault Tolerance. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Bordeaux, France, Article 8, 14 pages.

[19] Liliana Rosa, Luís Rodrigues, and Antónia Lopes. 2007. A Framework to Support Multiple Reconfiguration Strategies. In *Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems (Autonomics '07)*. Article 15, 10 pages.

[20] Fred B. Schneider. 1993. Distributed Systems (2Nd Ed.). ACM Press/Addison-Wesley Publishing Co., Chapter Replication Management Using the State-machine Approach, 169–197.

[21] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. San Francisco, California, 189–204.

[22] João Sousa and Alysson Bessani. 2012. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *Proceedings of the 2012 Ninth European Dependable Computing Conference (EDCC '12)*. Sibiu, Romania, 37–48.

[23] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. 2008. Support for Distributed Adaptations in Aspect-oriented Middleware. In *Proceedings of the 7th International Conference on Aspect-oriented Software Development (AOSD '08)*. Brussels, Belgium, 120–131.