Dynamic Adaptation of Byzantine Fault Tolerant Protocols

Carlos Eduardo Alves Carvalho
carlosacarvalho@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. State Machine Replication (SMR) is one of the fundamental strategies to preserve the consistency of a replicated system. Different protocols to implement SMR exist, each optimized for different operational conditions. In this report we study the algorithms that allow to replace, on-the-fly, one implementation of SMR by another SMR implementation, in scenarios that tolerate both crash faults and Byzantine faults. Our goal is to understand what are the techniques that allow to perform such dynamic adaptation with small impact on the system operation. Finally, we present an architecture to allow us to implement and compare, in practise, several of this techniques so that we can develop efficient solutions.

1 Introduction

Today, replicated distributed systems are widely used to increase the availability and performance of services provided in the Internet. Replication has the potential to improve performance, by distributing the load among different servers. Furthermore, replication can increase the system availability by allowing the service to be provided even if some processes fail.

One of the fundamental strategies to preserve the consistency of a replicated system is to use *state-machine replication* (SMR). This approach can be applied to any system that can be modelled as a deterministic state machine, a requirement that is met by a large variety of services like data stores, scientific calculators, document editors, etc. In simple terms, it consists on running the same application in multiple replicas, and use communication and coordination protocols to ensure that these replicas process the exact same sequence of requests (i.e., all requests are received by all replicas in the same total order).

The actual protocols that implement the communication and coordination support, such as consensus protocols[1], strongly depend on the properties of the system where the state-machine is deployed: the degree of synchrony of the infrastructure and the types of faults that can occur, are two examples of factors that affect the implementation of consensus. The more severe the faults we need to tolerate, the more complex and computationally expensive are the algorithms that implement consensus. In particular, algorithms that tolerate crash faults are

much more efficient than algorithms that tolerate arbitrary faults (also known as Byzantine faults[2]). Due to this reason, many systems only tolerate crash faults.

Byzantine faults may be caused by electromagnetic phenomena (such as bit flips) or be the result of malicious code (for instance, if a node in the system is compromised by hackers). Unfortunately, trends in hardware design (where gates are densely packed in a chip) and cybercrime, make the occurrence of Byzantine faults more likely. This has spurred the interest in developing efficient solution to support state-machine protocols that are tolerant to arbitrary faults, what is known as Byzantine fault-tolerance (BFT). As a result of this effort, several BFT protocols have been designed, such as PBFT[3] and Zyzzyva[4], among others. Despite all these efforts, there is not a single BFT protocol that outperforms all other protocols for all deployments. In fact, some protocols perform better in some conditions than others, for instance, it has been shown in [5] that PBFT first scales better with payload size while Zyzzyva provides greater throughput and is more robust in wide-area and lossy networks.

Given that the characteristics of the deployments where BFT systems operate may change often nowadays (as a result of both hardware upgrades and changes in the workload), it would be advantageous to leverage the optimizations of different solutions for different environment conditions. This way, instead of using a single SMR implementation, that is chosen at deployment time, and that may turn out not to be the most appropriate during a large portion of the execution time, it would be possible to always use the implementation that better fits the current conditions, at any given time. To achieve this in an efficient manner, it must be possible to adapt the system in runtime, with minimal interference on the provided service.

In this report we study the algorithms that allow to replace, on-the-fly, one implementation of SMR by another one, in scenarios that tolerate both crash faults and Byzantine faults. Our goal is to understand what are the techniques that allow to perform such dynamic adaptation with small impact on the system operation, such that it becomes feasible to react quickly to changes in the workload, the operational condition, or even in the level of threat for instance, by switching among crash-tolerant and Byzantine-Tolerant implementations.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we explore further the ideas behind SMR. Section 4 discusses the problems and some solutions to BFT SMR. In Section 5 we present some thoughts on protocol adaptation. Section 7 describes the proposed architecture to be implemented and Section 8 describes how we plan to evaluate our results. Finally, Section 9 presents the schedule of future work and Section 10 concludes the report.

2 Goals

This work addresses the problem of performing the dynamic adaptation of state machine replication protocols. More precisely:

Goals: We aim and identifying, implementing, and evaluating experimentally different techniques to reconfigure the SRM protocol, in order to understand which techniques are more efficient in systems subject to Byzantine faults.

To achieve the goal above, we will start by surveying related work, namely previous proposals to reconfigure SMR for both the crash and Byzantine fault-model. We will implement the most relevant of those techniques in a common framework, namely the BFT-SMaRt[6] framework and use those implementations to compare their performance, namely, their impact on the latency of ongoing request during the reconfiguration period.

Expected results: The work will produce i) a catalogue of different SMR reconfiguration techniques suitable for Byzatine fault-tolerance; ii) an implementation of each of these techniques in the BFT-SMaRt framework, iii) an extensive experimental evaluation using a cluster of workstations.

3 State Machine Replication

A general approach to provide fault-tolerance is to use State Machine Replication [7]. This approach sees the whole system as a finite state machine with a given *state* and that is capable of processing certain *commands*, which modify the state (deterministically). This state is replicated among all replicas of a given service and commands are executed in all of them. The rationale is to have several copies of a server, so if some fail, others can still provide service. It is common to have a *log* with the history of the requests processed by the machine. This serves two purposes, firstly to transfer state if a new replica is initiated (possibly to replace a faulty one). Secondly, to enforce synchronization among replicas. As faults may cause some deviations among replicas states, the history of each replica is used to, from time to time, decide upon a common history, so that the overall SMR keeps consistent. To decide on the order of the commands executed, there is the need for an agreement among the replicas, that is called *consensus*. A high level representation of the general architecture of a replica in SMR is presented in figure 1.

The replicas participating on a SMR system may change over time, because faulty replicas may be replaced by new ones, or the total number of replicas is changes (e.g. increased to withstand more faults). As it is essential that every replica knows about the others in order to carry out consensus, this state about how many and which replicas are participating is captured in a *view*.

SMR systems can be built to tolerate Crash or Byzantine faults, depending on the concerns while developing it, both approaches are discussed in more detail in the following sections (Crash in subsection 3.1 and Byzantine in section 4).

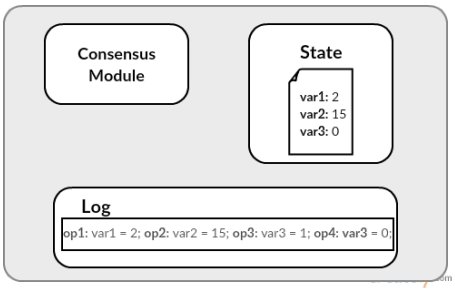

Fig. 1. The general architecture of a replica in State Machine Replication

3.1 Raft

Raft[8] is a protocol used to implement SMR in synchronous systems subject to crash faults. Raft can be seen as a more understandable consensus approach than the most known Paxos protocol, originally proposed by Leslie Lamport[9]. Raft requires at least $2f + 1$ replicas in order to tolerate up to f faults.

Raft is a *leader-based* protocol; one of the replicas is elected to play a special role and act as a coordinator for the protocol. The protocol includes the required mechanisms to replace a failed leader by another replica and to make sure that no inconsistency is generated even if, due to the asynchrony of the system, more than one replica believes to be the leader at a given point in time.

The leader replica receives requests from the client and broadcasts them to all the other replicas. Before a request is processed, the protocol ensures that it is ordered and added to the log of, at least, $f + 1$ replicas. When a given entry is appended to the log of a majority of the replicas, we say that it is *committed* and it is safe to answer to the client because, despite failures, that operation will always be part of the history of the state machine. Known to be committed, it can be executed and a response returned to the client.

If the leader fails, a new leader is elected. Each replica has a given time-out threshold to wait for messages from the leader. When a time-out occurs, the replica suspects that the leader may have crashed, so it initiates a leader election. Firstly it proposes itself as a candidate for leadership to all other replicas. Then, all replicas cast their vote, and if, and only if, a majority vote for it, it starts to be the leader and informs every other replica of its new role. Every time an election is started, it is started a new *term*, which is sent along all communications. Terms have increasing numbers and are used to enable replicas to know if other replica is further advanced or far behind, as the system progresses. As an example, if a candidate to leadership receives a request for processing a command from another leader and the term is greater than its own term, then it knows that the new leader already won an election further on the progress of the machine. Knowing this, the candidate stops the election, as it knows there was an election "further in the future" in which he did not participate, although a majority of replicas did. This can happen, for example, due to a transient fault

in the network, where some packets were dropped, and the replica missed some communications.

When a leader is elected, it is possible that inconsistencies of logs among the replicas may arise. As an example, the old leader may have entries in his log that were not fully replicated yet. To solve this, Raft forces all replicas to replicate the new leaders log. This is done by finding the latest point where the log of the replica is equal to the leader, removing further entries (that not exist in the leader's log), if they exist, and appending the missing entries that are present in the leader's log. Unfortunately, if no preventive measures are taken, the new leader can be a replica that failed to append some commands to his log. This could carry the risk of erasing parts of history which a client already knows of, and thus breaking correctness, because the overall state of the machine would not be consistent with the requests made by client. To solve this, Raft limits who can be elected as leader: only a candidate that has a log with, at least, the same entries as a majority of the replicas can be elected. This ensures that the new leader has all the committed entries, as by design, an entry is committed if a majority of the replicas have it in their logs.

As the main focus of this work is on BFT, we will not present more crash fault tolerant systems, as the on described earlier already provides a general view about the mechanics of crash fault tolerant SMR. In the next section we will present the problem of Byzantine fault tolerance, some systems and discuss the key differences between crash and Byzantine fault tolerant, as well the differences and similarities among BFT SMR approaches.

4 The Byzantine Generals Problem

The Byzantine fault tolerance problem was first described by Lamport, Shostak and Pease in [2]. The authors present the problem starting point as follows: there are several of the Byzantine army camped outside an enemy city, being each division commanded by a general. They must agree on how to perform the attack, because if they don't attack in accordance to each other they might face defeat. The problem arises because there are traitors among the generals that might try to impair this agreement so that the Byzantine army fails. Of course, this is a metaphor for a distributed where some machines may not act like specified or intended.

In the paper was presented a solution (among others) that became the starting point for most practical implementations of Byzantine fault-tolerant (BFT) SMR. The ultimate objective of the algorithm is to have all the non-faulty nodes agreeing on a value. These nodes are the replicas, when we talk about SMR. The value to be agreed upon is suggested by the leader, a node with this special role.

As discussed in the paper, the ability of faulty nodes to lie about the messages received from other nodes introduces difficulty in solving the problem. So as to mitigate this, the proposed solution uses signed messages to prove the source of a given message. This algorithm assumes a fully connected network, at most f

faulty nodes and at least $3f+1$ nodes. Below, we present an informal description of the solution:

1. Every node starts with a vector $V_i = \emptyset$, representing the collection of values received from other nodes.
2. The leader proposes a value v, sending a signed message to all other nodes.
3. Upon receiving a message each node acts accordingly with one of the next cases (in other cases the messages are ignored):
   A) If it receives a message from the commander for the first time, the node adds v to V_i. Then it signs the message and resends it to all other nodes.
   B) If the node receives a message with a value v', signed by $k+1$ nodes (including the commander) and $v' \notin V_i$, the node adds v' to V_i. Then if $k < f$, it signs the message and sends it to all nodes apart from those that already signed the said message, to try to guarantee that, even if f nodes fail, at least one correct node will know about the value propposed by the leader.
4. When a node will not receive more messages it chooses an action, based on a deterministic function, taking in account the values in V_i.

To know when a node will not receive further messages, it is needed to have some time-out mechanism. To achieve this, it is necessary to assume some maximum time for processing and transmitting a message, therefore a synchronous network needs to be assumed. Although this may seem a flaw, there is no possible solution for solving the problem under an asynchronous network assumption, as the FLP Impossiblity Theorem [10] proves.

4.1 Tangaroa

In this subsection we will present an adaptation of Raft that tolerates Byzantine faults, called Tangaroa [11]. We will compare both approaches so the differences between crash fault tolerant and Byzantine fault tolerant systems emerge.

The first key difference is that to tolerate Byzantine faults there is a need for more replicas, at least, $3f + 1$ [7], to tolerate f faults, as opposed to $2f + 1$ in crash fault tolerance [12].

Secondly, it is necessary to ensure authenticity in communications, as replicas may lie about messages received from other replicas. To enforce this, Tangaroa uses digital signatures. As an example, a leader could modify a command received from a client, tricking other replicas to execute something different from the real command and harming the safety of the system. If the client signs its messages, then it is theoretically impossible for a leader do tamper with it [13].

Thirdly, a Byzantine leader could starve the system by ignoring clients' requests while continuing to send heartbeats, this is, sending messages proving it has not crashed, although never broadcasting the requests. So there must be a mechanism that is able to detect this and act upon it. This approach solves it by allowing clients to denounce a leader if it does not answers to requests timely, so a leader change is triggered.

Furthermore, any replica could lie about its log, when asked to replicate the leader log, compromising the state of the machine. They could state that they had replicated it, but when in reality they have another sequence of requests in its log. So, further proof must be gathered to convince other replicas that a replica has, in fact, all the entries in its log. To prove this, a replica hashes its log and signs it, so others can compare the hashes with their own to check if the histories match.

Another issue arises because a replica could lie and elect himself as leader, without winning an election. To do this, a replica could just broadcast a message stating "I'm the new leader", making others look to it as the new leader. To prevent this, a recently elected leader, when informing the system of its new role, must sent cryptographic proof that a majority of replicas voted for him.

A Byzantine leader in Raft, as it is the single node that coordinates what is committed, could also tell that a given entry was committed even if a majority of the replicas had not appended it to their logs. By opposition, in Tangaroa, the responsibility of marking entries as committed is removed from the leader and belongs to all replicas. When a replica appends some entry to the log it broadcasts its action to all other replicas. The replicas collect these messages and identify by themselves that an entry was committed when a majority of replicas has, in fact, appended it.

Finally, in Raft a Byzantine replica could always be proposing elections, sending the system to a loop of infinite elections, where no progress is made. To mitigate this, in Tangaroa, a replica only casts a vote for a new leader if it also suspects that the leader is faulty. Otherwise, the election will be ignored, because all correct replicas will ignore the election proposal.

We can see that some overheads arise when a Byzantine fault tolerant approach is used. There is a need for more replicas, which some times is an issue, because makes the whole system more expensive. Moreover there is the need for producing cryptographic proofs to ensure authenticity, which causes a significant overhead in CPU and the time consumed to process communications, both when producing and verifying those proofs. Finally, a need for all-to-all communication arises, which increases the number of messages in the network exponentially, what can possibly introduce more latency on communications, as well, it demands further effort by the replicas to process all the communications. However, despite this extra overhead, sometimes a Byzantine environment must be assumed, when having a Byzantine failure is not tolerable, for example in critical control software.

Below we present and discuss some approaches for solving BFT SMR in practice.

4.2 PBFT

PBFT [3] is a widely-studied BFT algorithm that aims on solving BFT SMR in a practice. It was the first presented BFT protocol that relaxed the synchrony assumption, which is often not present in real world systems. It does not rely on synchrony to ensure safety, instead, it relies on the assumption that the network

has some times of synchrony to ensure liveness (assuring liveness and safety under total asynchrony is impossible [10]). Furthermore, there are presented some optimizations that try to reduce the response time.

As Tangaroa, this approach needs the theoretical minimum of replicas to work, which is $3f+1$ replicas, where f is the maximum number of faults tolerated. On a high-level view, the algorithm works as follows:

1. The client sends an operation request to the leader, that is responsible for ordering the operations on the system.
2. The leader atomically multicasts the request to the other replicas, called backups.
3. All the replicas process the request and answer to the client.
4. The client verifies if it received at least f+1 equal replies, if so it assumes that reply as the result of the operation.

In this system the view states what replica is the leader. When the leader is suspected to be faulty, a view change is carried out and another replica becomes the leader. This suspicion is arose when a backup notices that a request it knows of is taking too long to be executed.

The atomic multicast protocol is composed of three phases: pre-prepare, prepare and commit. The first two, together, guarantee that the requests are totally ordered within a view, even in the presence of a faulty leader. The latter two, in conjunction, guarantee that the ordering is kept among views.

In the pre-prepare phase the leader assigns a sequence number n to the request and sends a PRE-PREPARE message to all replicas. A backup accepts this message if:

- is in the same view has the leader,
- has not accepted a pre-prepare with the sequence number n for a different request, in that view,
- verifies the authenticity of the message.

If a backup accepts the PRE-PREPARE, it enters in the prepare phase. As it does so, it sends a PREPARE message to all other replicas. Every replica registers the PRE-PREPARE messages, its own PREPARE, as well as the PREPARE messages received from other replicas (as long as they have correct signatures and are in the same view).

When a replica has at least $2f$ PREPARE messages from different backups that are coherent with the PRE-PREPARE, it multicasts a COMMIT message to all other replicas. When a a replica receives at least $2f+1$ COMMIT messages matching the PRE-PREPARE, it processes the operation requested by the client and replies to it.

A view-change is initiated when a backup notices that a request is taking to long to be processed. To change the view, the replica stops participating in the processing of requests and multicasts a view-change message. When the leader of the new view gets $2f$ view-change messages, it informs all backups of the new view (with proof that $2f$ replicas agreed upon that).

In order to make the system faster in practice, among other optimizations, this paper introduces the usage of Message Authentication Code (MAC) instead of digital signatures to reduce the load on the CPU of the replicas. MACs are computationally less expensive because they use symmetric cryptography, as opposed to the asymmetric cryptography used to produce digital signatures, which needs more complex mathematical computations to be carried out.

4.3 Zyzzyva

Zyzzyva aims to make a fast BFT SMR system by using speculation. In this approach, a request is executed without running an agreement among the replicas to order it, in the hope that no failures occur, unlike other systems such as PBFT. Logically, to ensure safety even under the presence of Byzantine faults, an agreement is run when suspicions of faulty behaviour occur. To detect this type of behaviour, the client is responsible to detect and inform about some divergence of answers received by it. As PBFT, Zyzzyva relies on a leader (called leader in PBFT) to give a sequence number to the requests and uses $3f + 1$ replicas (f being the number of tolerated faults).

In the fast case, where there are no faults nor divergence on the state of the replicas, the protocol works as follows:

1. The client sends a request to the leader.
2. The leader gives a sequence number to the request and forwards it to all the replicas.
3. Each replica executes (speculatively) the request and sends the response to the client. If the client receives $3f + 1$ equal answers, it is safe to rely on the answer because, at least all the correct replicas, will keep this request consistently ordered in their history.

On the other hand, if the client receives only between $2f+1$ and $3f$ matching answers, further steps must be taken to ensure safety. It may indicate that the state of the replicas is diverging, due to faults in them or the network, because some there is no proof that all processed the request the same way. Therefore, the consistency of the general state of the system may be at risk of becoming inconsistent when a view change happens. This could happen, for example, under the presence of a faulty leader, that could orchestrate an attack by lying to some correct replicas, making it possible to agree on a faulty state when a view-change happens. To prevent this kind of faulty behaviour to harm the system's correctness, the client builds and distributes a certificate that proves that at least $2f + 1$ replicas agree on the answer of the request with a given sequence number, so when a view change happens, it is ensured that there is proof that a quorum of $2f + 1$ replicas agreed on the order of the request in the past. To ensure that enough servers received this proof, it awaits the acknowledgement of at least $2f + 1$ replicas. This amount of replicas ensures that even the presence of f faulty nodes that will lie about this acknowledgement, there is a sufficient number of replicas ($f + 1$) to prove that this agreement existed.

If the client receives less than $2f+1$ matching responses, which is not enough to ensure that a majority of correct replicas agreed on some response, it suspects a faulty leader and resends the request to all replicas. If a given replica has not processed that request already, it forwards the request to the leader. If, after a certain amount of time, it has not received the corresponding request ordered by the leader, the replica starts a view-change.

The usage of speculation and having an agreement protocol with just two steps introduces extra complexity on the view-change protocol. Having just two phases, omitting a phase when the replicas share their state, makes the traditional view-change protocol unsafe, as the correct replicas might not be able to initiate a view-change in the presence of a faulty leader (it is possible that only f correct replicas notice it). To ensure that at least $2f$ replicas commit to a view-change, there is a need for an extra step in the view-change protocol, where the replicas state their disagreement about the leader. So, the protocol works as follows:

1. A replica informs all other about is suspicion about the leader.
2. Upon receiving $f+1$ confirmations of the suspicion on the leader from other replicas, it is created a proof that at least one correct replica suspects the leader and it is sent along a view-change message. All correct replicas, given this proof, will commit also to the view-change.
3. The leader of the new view collects $2f+1$ view-change messages and then send a new-view message with proof of that.
4. Finally, when a replica receives the new-view, it changes its view.

4.4 Aardvark

Aardvark [14] addresses the problem of BFT SMR with a very different mindset of the previous solutions. The authors reject every optimization that could impair the performance in cases where faults happen. This way, Aardvark maintains a steady throughput, even in the presence of faults, making it more robust w.r.t. Byzantine behaviour, in the replicas and the clients. This system as a similar communication pattern to PBFT, using a leader to sequence clients' requests and a three phase agreement protocol, although it presents some key differences on the implementation.

Firstly, to minimize the harm a Byzantine client could make on the system, clients digitally sign their requests, instead of using MAC (although, MAC are also used to optimize in some cases). This way it is ensured that if a replica can prove the authenticity of a request, all other can too, because signatures provide non-repudiation, so when a client signs a message, there is (theoretically) undeniable proof that it indeed sent that message. Although signatures are more expensive to compute than MAC, this simplifies the algorithm, removing some corner cases found on other systems. For example, the authors found out that in PBFT and Zyzzyva if a client would send a request with a valid MAC for the primary and invalid MACs for the other replicas, it would render the system unusable. PBFT would incur in recurring view changes, while Zyzzyva would invoke

a conflict resolution protocol that in practise, due to not be fully implemented, would never finish.

Secondly, also to mitigate the impact of faulty clients, it uses separate queues for messages from the clients and replica-to-replica communication. This way, a replica can guarantee that only a portion of the resources can be used by the client, ensuring that a client can not stop a replica from making progress in requests already received by flooding the network. This approach also uses independent network interface controllers and wires to link each pair of replicas. So, at the expense of having to rely on point to-point communication, it is possible to receive messages in parallel and also shut down links to faulty nodes trying to develop a denial of service attack.

Lastly, Aardvark carries out view-changes regularly, as opposed to carrying it only as a last-resort measure. The authors justify this decision stating that the cost of having a faulty leader surpasses the cost of executing periodic view-changes. In order to carry the changes periodically, the replicas demand an increasing minimum throughput of requests by the leader, as soon as the leader fails to provide such throughput, the replicas start a view-change.

4.5 Discussion

When comparing the machinery needed to tolerate Byzantine faults versus the mechanisms needed to tolerate only crash faults, it is evident that the first carries significant overheads. Firstly there is a need for, at least, more f replicas to tolerate f faults. Secondly, all-to-all communication introduces much more load in the network, which can cause starvation. Moreover, the use of authenticated messages introduces more load on the CPU to process the cryptography, as increases the time needed to process each message. Nonetheless, this all-to-all communication, with a heavy CPU load to calculate authentication proofs, can be avoided in some cases as show by Chain in [15]. In Chain all replicas form a chain and a request is transferred from node to node in the chain, reducing the cost of all-to-all authenticated communication, but introducing extra latency, because the messages are processed sequentially in every nodes, instead of in parallel. However, as stated earlier, these overheads in latency, network resources and CPU, could pay off if the gain in resilience and availability is essential to the service provided.

The three practical BFT SMR protocols presented above show us that different approaches and optimizations can be taken in order to produce BFT systems. Each of the approaches carries a gain in performance some cases while sacrificing some of it in other situations. For example, when no faults occur and the network is stable, Zyzzyva has the best performance, due to its fast and two-phase cases. On the other hand, under the presence of faulty clients, Aardvark beats Zyzzyva due to its resilient design. A faulty client can also harm the performance in PBFT, sending inconsistent MAC authenticators, one for the client and other for the other replicas, this would send the system to recurrent view changes, limiting its progress [14]. Moreover, when comparing PBFT with Zyzzyva, we

can denote that the first is more predictable and steady performance under increasing payload sizes of the requests. On the other hand, Zyzzyva offers a better performance in wide-area network, where packet loss is frequent [5].

It's also noticeable that, despite their major differences, all the approaches can be decomposed in the same modules. All implementations have a protocol for the agreement and a module for view change. The clients have also different behaviours and responsibilities, but in all of the systems they all request something and act upon a reply (or the absence of it). Moreover, the view carries a slightly different meaning and information with it in the different approaches, but, then again, they are always responsible to capture the configuration and (to some extent) the state of the system in a given timespan. So, a pattern starts to emerge when we look to this algorithms from a higher level, which is very important to be able to modularize these in order to adapt them, specially if we are looking for a general solution that can work with arbitrary algorithms.

5 Approaches on Protocol Adaptation

There is a wide spectrum of adaptation that can be made to protocols, that can affect different ranges of replicas, as well as they can demand distinct levels of consistency on coordination. Adaptations can have effect on all replicas or in just a subset, as an example, a change in the agreement protocol must be known by all replicas, while a change in the batch size, usually, only concerns the leader. Moreover, adaptations may demand an atomic agreement (no request can be processed with different settings among the replicas), for example, if the authentication method changes, all replicas must know it before processing any further request, otherwise inconsistencies may be introduce in the system. On the other hand, if it is the time-out threshold to detect replicas failing to answer that changes, most of the times is not paramount that it happens at all the replicas simultaneously to keep the correctness of the system. Below we present some adaptations that can be made for each of the categories.

- **Adaptations that have effect only on a subset of replicas:** Change in the batch size of requests, changes in logging techniques, etc.
- **Adaptations that have effect in all replicas:**
  - **Demanding an atomic agreement:** Changing the underlying agreement protocol; changing the BFT SMR approach; changing the authentication method used for the messages.
  - **Demanding eventual agreement:** Changing the time-out threshold to detect replicas failing to reply;

To carry out such configurations is then needed an orchestration between the replicas of the system, if the adaptation affects more than one replica, and some local reconfiguration strategy [16]. The orchestration component is responsible to coordinate the reconfiguration among the replicas in order to ensure the correct function of the system. This orchestration works much like a synchronization barrier, so that all the replicas move through the reconfiguration steps in sync

with each other. This reconfiguration steps are the local reconfiguration strategy, that defines how the local steps needed to change the configuration.

The combination of local strategies with different kinds of orchestrations can provide several approaches on reconfiguration, each one has different consistency guarantees and different performance [16]:

- **Flash:** In this strategy all the replicas apply the changes locally, without care for other replicas. This is a strategy that introduces little delay during a reconfiguration, although only works for adaptations that do not demand for an atomic agreement.
- **Interrupting:** This strategy stops the systems, applies the new configuration and only then starts the system again. This way and atomic agreement for when the new configuration is applied is guaranteed, but the delay introduced is considerably bigger than the previous solution.
- **Non Interrupting:** To try to minimize the delay of an interrupting strategy while maintaining the guarantees of an atomic agreement adaptation, this approach runs both configurations (old and new) simultaneously until the new one is fully functional, then the old can be shut down. Nevertheless, this introduces more complexity in the orchestration than the previous solutions, it also carries the computational overhead of having two configurations running at once.

5.1 Adaptation of SMR Protocols

Any adaptation strategy used may ensure the properties of systems that is being adapted. When talking about SMR usually we want to ensure that every request is totally ordered at most once, thus being necessary to ensure that a given request that was already ordered previously to an adaptations is not ordered again.

Another key property of SMR that needs to be ensured during and adaptation is *irrevocability*. This property guarantees that if the systems outputs to the user that some request was executed, then the state of the system must always reflect that execution, even if some processes fail. More informally, if a user successfully executed an action, like depositing some amount of money, then it as assurance that in the future the money will still be in her account, despite what failures can happen, even if the machine where the money was deposited explodes. SMR offers this because otherwise it would be fuzzy to define even the semantics of tolerating a fault. So, when adapting the system we must assure that no request is dropped from the history if a client already knows about its execution.

5.2 Configuration as a dynamic module

When talking about adaptations in the context of SMR, we can use its own consensus module to execute the orchestration because the synchronization can be made be deciding on the steps to perform. Lamport, Malkhi and Zhou[17] presented a method to produce algorithms for adaptation that uses the inter-replica

agreement naturally present in SMR systems to decide on a new configuration, using the already existing interface to propose commands. The agreement is dependent on the configuration, so, to agree about a request i, all the processes must be using the same configuration. So, to allow the processes to know how to behave, a specification of the current configuration is kept in the system's state. To change it, there must be an agreement, so a new special request is introduced, RECONFIGURE(C), which specifies the new configuration C. So, when RECONFIGURE(C) is agreed upon as the request i, the configuration is set to C, so from request $i+1$ and on this new configuration is used. This method is called by the authors R_1.

Nevertheless, if we aim to adapt the underlying protocol, developing a system like this in practise would have some drawbacks. Firstly, it would be a complex monolithic algorithm, due to be a composition of usually already complex SMR algorithms, so it would be harder to develop and to prove correct than smaller non-adaptive algorithms. On the other hand, it would be harder to extend to keep up with the state of the art, probably becoming obsolete in a short time. An alternative to this is using black-box switching.

5.3 Black-Box Switching

We designate by black box-switching the task of building a reconfigurable state-machine from two state-machine implementations that have no support for reconfiguration, not even any special command to put the state-machine in a quiescent state.

In this context we say that a state-machine is reconfigurable if it accepts a special command RECONFIGURE(C, C') that can be applied to configuration C to change the state-machine to configuration C'. Since one aims at providing this abstraction using state-machines that have no support for reconfiguration, the solution consists in instantiating two different state-machines, a state machine $S1$ running configuration C and another state machine $S2$ running configuration C' and, at some point, start redirecting all request to the second state-machine.

There are two main challenges in this approach. The first is how to know that is safe to stop using $S1$ and start using $S2$. The second is to avoid a long hiatus, where no commands are processed, during the switching operation.

A naive look at the first problem could indicate that a simple, yet not efficient, solution to the first problem would be to coordinate all nodes to stop submitting commands to machine $S1$. When one is sure that new commands are no longer being submitted to $S1$, one would simply wait for all commands previously submitted are ordered, and then one could resume the operation by submitting new commands to $S2$. Unfortunately, this strategy is only feasible in a system with a perfect failure detector. In the general case, it may be hard to ensure that clients and replicas that are not reachable have reached a quiescent state.

Due to the problem above, most solutions rely on using state machine $S1$ to define which is the last command to be ordered by $S1$. This is implemented by issuing a special command that works as a marker. All commands that are ordered after the marker can no longer be processed and need to be resubmitted

to state machine $S2$. Note that this approach does not require $S1$ to be made quiescent. $S1$ may still process commands after the marker, but the results are ignored to prevent the duplicate execution of commands, so these need to be re-processed by $S2$.

The approach above solves the first problem but does not address the efficiency problem. In fact, a large number of commands may be affected by the reconfiguration, being ordered after the marked and needed to be re-submitted (to be re-ordered again) to machine $S2$ which may double the latency of command processing during the reconfiguration procedure. In the next paragraphs we discuss some approaches to mitigate this problem.

5.3.1 R_α Lamport et al. presented an improvement to R_1 so that it could deal with parallel commands being decided after a reconfiguration happened. This is, in the R_1 approach, if the agreement $i+1$ was being agreed upon when a reconfiguration was decided, at command i, $i+1$ would have to be resubmitted to the new configuration, as after the reconfiguration no more decided commands should be processed to ensure that it is not processed by both the old and the new configuration.

In order to allow concurrent processing of requests, the state machine must then delay the change of the reconfiguration when a RECONFIGURE(C) is agreed on. This is, if a RECONFIGURE(C) is decided as the request i, the reconfiguration only takes place after executing the request $i + \alpha - 1$, being α the maximum number of concurrent agreements running at a giving time. This allows for the requests that are being agreed upon in parallel with theRECONFIGURE(C) to use the old configuration safely, because it is ensured that a reconfiguration request will only affect processes that are not being decided concurrently. This method is called R_α.

If it is necessary to make the reconfiguration take place immediately after a RECONFIGURE(C), this is, no request will be executed between deciding RECONFIGURE(C) and the configuration taking place, $\alpha - 1$ *noop* commands must proposed right after the RECONFIGURE(C). This can be batched in order to consume the same resources as it was only one command.

However, this solution would imply buffering all the commands that arrive during the reconfiguration time, so that they are then processed by the new configuration. This would then cause an interruption on the processing of new requests and a degradation in the quality of service due to a drop in the throughput of operations done.

5.3.2 Run-time switching between algorithms Mocito and Rodrigues [18] try to mitigate the problem of interrupting the processing of new messages during an adaptation event by using an approach similar to the non interrupting approach presented by Rosa et al. [16]. Despite of the work of Mocito et al. was focused on changing between total order algorithms, we believe that the ideas present in it can be adapted to SMR, as total order is recurrently a central piece in the development of SMR systems. The main idea of the work of Mocito *et aç.*

is to have both algorithms, the one that is currently running and its successor, running simultaneously during the switching time (since the adaptation command is issued until the new configuration is fully functional). On a high level view, to switch from algorithm A to algorithm B, the protocol works as follows:

1. A special message stating the intended switch is broadcast to all processes.
2. When a process receives this message, it starts using both protocols and sends a flagged message notifying this. Although, until the switch is on the final stage, only algorithm A messages are delivered. Algorithm B messages are buffered in order.
3. When every process is using algorithm B, algorithm A is stopped. Then, all messages from B that are buffered and were not delivered by A are delivered in order. Finally, B starts operating as normal.

This approach effectively eliminates the downtime in service that would happen if a algorithm was stopped and then the new one was initialized. On the other hand, this solution may introduce a significant overhead on network if the protocols individually operate near the limit of the available bandwidth. This happens because the network, in order to not become a bottleneck, must support both algorithms running at the same time. Despite this, in the paper is discussed an optimization to mitigate this issue that consists in sending only the headers of the current algorithm (A) during the switch. So, this way, the payload is only transmitted using the new algorithm (B), reducing the load on the network.

This solution leaves yet two problems to solve, it does not allow for concurrent reconfigurations and it relies on a perfect failure detector. The first problem is not really a concern to us, as we assume that the adaptation manager present in the Abyss system will not send concurrent reconfiguration commands, as it would be a source of overhead. The second issue is of major importance to us, as we want to develop fault tolerant systems, we can not rely on a perfect failure detector. In case of a replica crashing or getting mute, an switch would never finish because there would be a replica never stating that it was already using the new algorithm.

5.3.3 Building a reconfigurable state machine from non-reconfigurable ones

Bortnikov et al. [19] further explored this black-box approach, mitigating the problem of having to resubmit some requests to the new configuration, like Mocito et al., but under a crash fault tolerant paradigm. So, the authors of the paper present a framework to develop a configurable state machine from non-configurable ones, assuming only reliable communications (messages sent from a process to another are eventually received) and a crash fault model. The work focuses on the change of the set of replicas participating on the state machine execution at a given point in time, this is, switching between non-reconfigurable SMR implementations that work with a fixed number of replicas. Although, this work can be extended to support the switch between SMR implementations that not only can have different number of replicas, but also can execute in different ways, having distinct patterns of communication, for example.

In this approach, the processes of the reconfigurable state machine must be able to state that they are ready to start processing requests under the new configuration. This is, upon RECONFIGURE(C'), the processes broadcast READY(C') as soon as they are aware of it. When any replica receives READY(C') from a quorum of replicas, then they start using the new configuration and state to all active processes (under all configurations) that they changed its own configuration with a NEW-CONF(C') message.

Switching between different state machines generally carries the overhead of transferring the state from one to another, in order to ensure total order. To mitigate this overhead, this work is built on top of the premise that different configurations (i.e non-reconfigurable state machines) must be independent of each other, so they can start operating from the initial state, independently of the history of the previous configurations.

Another optimization introduced, leveraging the independence of the configurations, is the concurrent speculative execution of new configurations, as soon as they are proposed, even if not already decided by the current configuration. This way there is no need for resubmitting requests to a new state machine, as every request that happened during the switching event was already submitted to it. Although, a problem may arise if concurrent reconfiguration changes occur, it would form a tree of commands and configurations (as in figure 2), instead of a single ordering line, thus breaking the total ordering of commands. To solve this, all replicas prune this tree in a deterministic way, by choosing always the first reconfiguration decided to be the one that is kept in the global ordering as the next. To inform this decision to the configurations that are being executed speculatively, the state of each configuration is shared periodically among the configurations, so they can prune the tree accordingly.

Although the dealing with concurrent reconfiguration requests is not a concern for our work, this speculative approach reduces the delays of deciding and starting a new configurations, specially under high-latency networks, mitigating also the need to buffer or resubmit commands that were being processed during the reconfigurations. To ensure that there are no commands duplicated by executing speculatively, the reconfigurable state machine only proposes new commands to the currently operating state-machine configuration if no reconfiguration was already decided by that configuration. If some reconfiguration was already decided, then, by design, some other state-machine implementation is already responsible for ordering such commands,

To use this SMR algorithm in practise there is a need an additional component, the Command Queue (CM). This component is responsible for associating the clients' commands with configurations, proposing it in the running configuration or configurations, if concurrent speculative configurations are executing. So, this component must be aware of the state of the active configurations in the system, which does by keeping track of the *ready* messages, to know which configurations are running at the time.

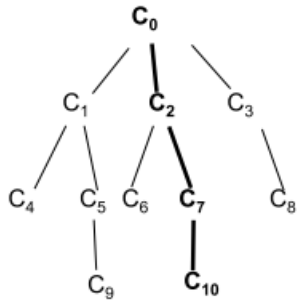

Fig. 2. A tree of commands and configurations resulting of speculative execution in Bortnikov *et al.* reconfigurable state machine approach. The edges correspond to the history of commands executed between two configurations. The highlighted path corresponds to the history kept in the global history, being the other discarded at some point.

5.4 Stopping adaptation

Despite the black-box approach was already proven as a feasible solution, it would be easier if the different state machines had a *stop* primitive to put it in a quiescent state. This way we can be sure that at some point in time a given configuration has stopped, having a giving final state, and it would not process further requests. This would allow for a simpler management of the adaptable system, even under a fault-tolerant paradigm, as by design, it would be possible to know when a machine has stopped and when it would be safe to send new requests to other state machine.

The basic idea behind this stopping kind of adaptation is to send a stop-sign to a given state machine, and then the state machine stops executing requests. All the requests addressed to the machine will receive an answer stating that the machine has stopped, and eventually some kind of pointer to the new one [17]. Once again, if multiple agreements are run in parallel further care must be taken. The stop-sign would impair sending an execution guarantee to the client as soon as the request is agreed upon, it would have to wait for all previous requests to be executed, as the machine could stop before. In this case, like the $R\alpha$, a delayed stop-sign must be used, so the stoppage happens after α agreement instances, guaranteeing that no execution guarantee is violated.

In the same paper is also discussed another way to stop a machine, sending infinite noop requests. This derives from the conceptual thinking of a machine execution consisting in some finite non-noop requests followed by infinite noops. This can be easily represented in finite manner, while using batching.

5.4.1 The next 700 BFT protocols These ideas of stopping the running replicas and spawning new ones with a different configuration were explored with a BFT mindset in Abstract [15]. The main idea is the development of several

components, called Abstract instances, that, at a given time, one instance is running and providing service to the client, when some event of interest happens, an adaptation event occurs and other Abstract instance takes its place. This adaptations event is a generalizations of a *stop-sign*, as it is a deterministic condition checked by each replica themselves, it can be a *stop-sign* sent by another process (e.g. the client) or any execution or environmental condition, as some time-out or deciding a given amount of commands to be executed. Moreover, this solution deviates from other BFT SMR, like PBFT and Zyzzyva, as it may abort client's requests.

More concretely, an Abstract instance implements a BFT protocol specialized for the given system conditions, called progress conditions, as it only needs to guarantee progress under that conditions. If progress conditions are not met (i.e some assumed condition fails), the Abstract instance aborts, and other with weaker system assumptions takes its place. Therefore, to guarantee the correct functioning of the whole system, it is necessary to guarantee that no request is aborted by all instances. As an example, it is possible to have an Abstract instance that only makes progress when there are no faults, aborting otherwise, in this case it is necessary to ensure that some other instance is capable of dealing with faults, or a client would never get its request answered. As suggested by the authors, this is usually achieved by using a robust well-studied BFT algorithm as one of the Abstract instances.

The transition between instances is mediated by the clients, as they propagate the history of an aborted Abstract instance to the next one. So clients need to be aware of the adaptations happening in the server replicas, not only for interacting with them correctly, but also to carry the state. The transition happens in three steps: stopping the current instance, choose a new one and finally initializing the chosen new one. As choosing what adaptation to do is out of the scope of this work, so next we will describe in more detail only how an instance is stopped and initialized:

1. **Stopping the current Abstract instance:** An Abstract instance stops when it aborts the first client request, due to the violation of the progress conditions. Along with the abort notification, an *abort history* is also sent. The mentioned history contains, as prefix, the *commit history* of the instance, and possibly some uncommitted requests.
2. **Initializing the new Abstract instance:** The client invokes the new Abstract instance with the *abort history* of the previous instance. This history is used to define the initial state of the instance, before it starts processing new requests. Abstract does not need any explicit agreement to decide the one common *abort history* among the replicas of an aborting instance. This is possible because , by design, every *abort history* as the same *commit history* as prefix, being this enough to make possible the guarantee of total order.

If switching through a client is a problem, in the context of some specific system, it is possible to extend Abstract to allow switching through a component responsible for the reconfiguration, or even to do the switch through the replicas. In the first case, an aborting Abstract instance must send the abort

notification and the *abort history* to the dedicated reconfiguration component, and this component is responsible to invoke the new instance with the history received. The latter case, switching through replicas, is possible by making each replica act as a client, being able to send a *noop* command, it would mediate the instance switching without making any modification on the state of the system.

Having this switching mechanism through the client, without any preemptive action to start new Abstract instances could carry the weight of making a client wait for the new instance to initialize, as opposed to the solution of Bortnikov et al.

Abstract presents itself as powerful solution to facilitate the development of adaptive BFT systems. It allows for better system qualities (e.g. throughput, latency) by allowing the use of specialized protocols for each situation. On the other hand, it alleviates the difficulty of developing BFT systems, by making possible the development of several simpler modules, that as a whole implement a full BFT system.

5.5 Further considerations

The problem of adapting algorithms in run-time is recurrent under different contexts. In this section we will explore some interesting ideas on adaptations on systems that were no developed to adapt SMR systems, but have some interesting considerations abou adapting distributed systems in general, which can be used in the BFT SMR context.

5.5.1 Multi-step algorithm switching Chen, Hiltunen and Schlischting explored these ideas of non-stopping adaptation, and developed an architecture and method to build a distributed system that is able to adapt between configurations gracefully [20]. In this approach, every adaptable component (e.g. total order component) has several *adaptation-aware algorithm modules* (AAM) and a *component adaptor module* (CAM). The first type of module is responsible to provide some specific implementation for the module functionality. The CAM is responsible for managing the adaptation of the module, this includes switching safely between distinct AAMs, coordinating the change with the other replicas in the system and detecting when and which adaptation should occur.

Generally, switching between algorithms in a distributed system must take into account the messages flowing in the network, as they may end up being received by a replica operating on a new protocol that does not recognize it. The authors of the paper try to solve this in a seamless manner by using a three-step change algorithm. Upon a adaptation event, the component prepares to receive messages from the new AAM, as well as control messages for the switch-over process. When all components across all replicas are prepared, they start processing the outgoing messages with the new AAM. Finally, when all replicas are sending messages with the new AAM, the component can stop receiving messages from the old AAM. The AAM must be aware of the adaptation process, as it must provide an API to execute the three steps described earlier. As both modules are

active during the switch-over, incoming messages must be identifiable as being from the old AAM or the new, this easily achievable by using specific headers or a similar mechanism.

This approach presents itself has a modular and understandable approach for reconfiguring a distributed system seamlessly, without stopping the whole system in order to carry out change, being more general than Bortnikov's *et al.* approach and introducing less load on the network than the solution of Mocito *et al.*. Nevertheless, it demands the algorithms used in adaptable components to have awareness of the adaptation process. This way, the developer must adapt the algorithms, what can be a drawback, because understanding some algorithms implementation can be a hard and time-consuming task.

5.5.2 General *vs* specialized adaptors Couceiro explored the use of a general stopping approach *versus* the use of specialized, custom built, non-stopping switching mechanisms [21]. In this work, the authors implemented a framework to allow the adaptation of protocols in the context of replicated in-memory transactional systems. This framework allowed to have different reconfiguration protocols to switch between algorithms, having a general "stop and go" approach, as well as being able to support custom tailored mechanisms to switch between two particular algorithms.

The "stop and go" approach described by the authors demanded a given protocol for replicated transactional systems to have a *boot()* primitive, that initializes the said protocol from inactivity, and a *stop()* primitive that stops the protocol, putting it in a quiescent state. On the other hand, the specialized switching algorithms described take advantage of the particularities of the algorithms. The solution presented used a similar approach to the one of Mocito et al. [18], with both algorithms running at the same time during the transition. Although, it deviates from it because how and when each protocol really processes requests is decided by the switching algorithm itself, so, they may be running in parallel, but only one processing requests, for example.

A practical comparison between both approaches is presented in the paper, and, as expected, the drop in performance when executing a custom switch is much smaller than when compared with the general approach. This happens because the specific approach leverages the white-box approach of the algorithms to perform a switch without having to interrupt the processing of new requests, or with very little interruption. The big downside of these specialized approaches is that they demand for a deep understanding of the specifics of the algorithms to be switched. Moreover, switching between some algorithms can be a very hard problem to solve, if not impossible, due to incompatibilities that often occur between different approaches.

5.6 Discussion

Building adaptive SMR systems switching among other (static) SMR implementations as black-boxes seems a more feasible approach then developing a

monolithic, complex and hard to manage and extend intrinsically adaptable SMR algorithm. Nevertheless, this is done ate the cost of building an adaptation coordination capable of performing the switch among two different state machines efficiently while maintaining the correctness of the overall system. To ease this coordination one could have the different state machines to be stoppable, this is, being a stopping primitive to put them in a quiescent state, where it is guaranteed to make no further progress. Nevertheless, this implies the development of such kind of algorithms, either from scratch or adapting already existent ones, which can be a complex and costly task which a practitioner may not want to endure. As an example, if we analyse the differences between Paxos [9] and a stoppable version of it [22], we can see that deriving stoppable algorithms may be not as trivial as it seems upon a first look.

When talking about adaptation of the underlying protocol in a SMR systems, the client usually needs to be somehow aware of the adaptations that are happening in the server replicas, because, as discussed earlier, the client can play different roles in different protocols, specially in BFT SMR. Messages flowing in the network may also have incompatible formats, which can introduce further challenges. As an example, if we use a black-box specific SMR implementation as a configurable parameter of our adaptive the state-machine, switching between a protocol that uses symmetric cryptography to another that uses signatures is not trivial because messages already sent by clients in the older protocol can not be processed in the new one. Given this, either the client gets its request refused and must resend in a new message, much like the stop-sign approach, or the message must be translated, which can be hard as the private key of the client is secret. Therefore, we can denote that both non-stopping and stopping approaches get very similar in this case. So, we can denote that the optimizations introduced by concurrently or speculatively ordering new commands under a new configuration may not be useful to its full extent, as the client must resend its request with a new format anyway.

In summary, both approaches, black-box and using stoppable algorithms, are different paths to achieve a goal but, due to optimizations and implementational details they can produce similar algorithms, as stated in [17]. So, discussing if a non-stopping approach performs better than a stopping approach is complicated because there is no comprehensive study comparing the two. Although we think that the more specialized approach we take when doing a reconfiguration, the least overhead will be introduced [21]. Therefore, in this work we will try to implement several approaches, more general and more specialized, in order to compare them in practise.

Deciding which configuration to use and when to do it is also a concern when talking about adaptive systems. However, that is out of the scope of this work, as it is part of the Abyss project, where these decisions will be made by other components.

6 BFT-SMaRt

There are available several open-source libraries to ease the development of BFT SMR systems, namely UpRight [23], Archistar[24] and BFT-SMaRt[6]. The first was the first attempt, that we are aware of, of a library concerned about simplifying the development of BFT systems. Although, these concerns introduced a significant overhead in the performance of the system [6]. Moreover, as of the date of the writing of this work, the library seems to not be maintained any more, being the last release dated of the 27 of January of 2010. Archistar is a compact BFT replication engine, although it is not easily extensible, because it is a monolithic approach. As UpRight it is not currently maintained, and, moreover, has very little documentation, which can make implementation problems harder to solve, in the future.

Given this, in this section we will present BFT-SMaRt, which will be the starting point of our work, because it is still maintained and it is highly modular, facilitating its extension. Besides this, some members involved in the Abyss project have deep knowledge about it, making it easier to solve implementation challenges, if they arise.

This library, implemented in Java, implements a BFT system similar to PBFT. Although it deviates from PBFT, as it uses a modular approach, instead of a monolithic one, to develop something more understandable and extensible. To develop an application using BFT smart, a developer needs only to implement the usual *invoke(command)* on the client, and an *execute(command)* on the server side, leaving all the responsibilities to ensure BFT to the library itself. Besides this, if more complex behaviour is needed, BFT-SMaRt can be extended using plugins, alternative calls and call-backs.

BFT-SMaRt is also able to add or remove replicas form a given system, carrying out the state-transfer needed to initialize the new replicas. This state capture and transfer is isolated in a layer between the replication protocol and the application, so it does not influence the consensus algorithm. To carry out such tasks three principles are used: logging the operations executed in the system, taking snapshots of the progression of the system, in different points in time at different replicas (so the system does not stop) and, finally, transferring the state to fresh replicas in a collaborative fashion, with distinct replicas sending different chunks of the state to the initializing replica.

The architecture of BFT-SMaRt, depicted in Fig. 3, has the following main modules:

- Extensible State Machine Replication: responsible for implementing that application.
- Mod-SMaRt and VP-Consensus [25]: responsible for implementing the SMR mechanics, including total ordering.
- Reconfiguration Module: responsible for carrying out the addition or the removal of replicas.
- State Transfer Module: responsible for initializing new replicas with the current state, or even recovering the whole system.

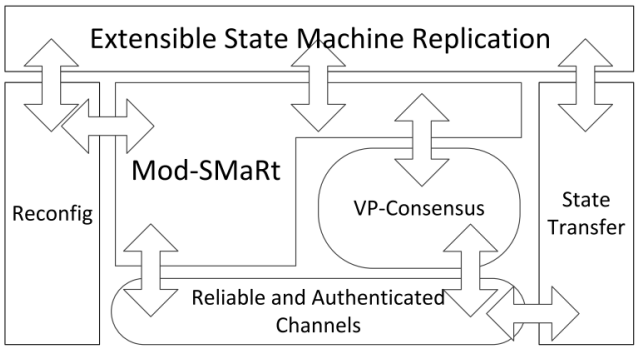

Fig. 3. The architecture of BFT-SMaRt

BFT-SMaRt was used in a work already done in the area of BFT protocol adaptation [26]. In this work, Sabino, developed an architecture, based on BFT-SMaRt, to develop adaptable BFT SMR systems, including a component to monitor the environment of the system and an adaptation manager, that reacts upon the happening of certain events, like a change in the network conditions. Despite this, the adaptations explored in this work are changes in protocol parameters like the batch size and the number of replicas, not discussing the main concern of our work, protocol commutation.

7 Architecture

We will developing a modular architecture able to support both black-box and stopping algorithm approaches, where where several implementations of BFT SMR will be seen as a configuration parameter of the system. This way it will be possible to implement solutions using both approaches in order to compare and rationalize about them. Nevertheless, the main goal is to find a strategy to switch between this algorithms introducing as little overhead as possible, not only during the switch, but also in the normal operation.

The system will be built on top of the BFT-SMaRt architecture, where the Mod-SMaRt (and VP-Consensus) will be replaced by a module capable of switching between different algorithms on-the-fly. Remember that the Mod-SMaRt is the module responsible for implementing the state machine behaviour and communication pattern between its replicas, this is, roughly, total-ordering the requests and maintain the state consistency. We will leverage the modularity of BFT-SMaRt with regard to having the different algorithms sharing the same replica state, so the need for state transfer between two algorithms upon an adaptation is reduced.

Unlike solutions like [20] or Abstract, the adaptation manager will not be responsible to know what neither when to adapt. As this work is part of the Abyss project, this responsibility is relied upon another system, which is being

developed in parallel. So, the focus of this solution is on how to perform adaptations suggested by another system, specially on how to switch between different SMR protocols. We assume that if a configuration is suggested by a configuration proposer component, then it the same configuration is proposed to all of the replicas of the system. A view of the architecture is described in Figure 4.

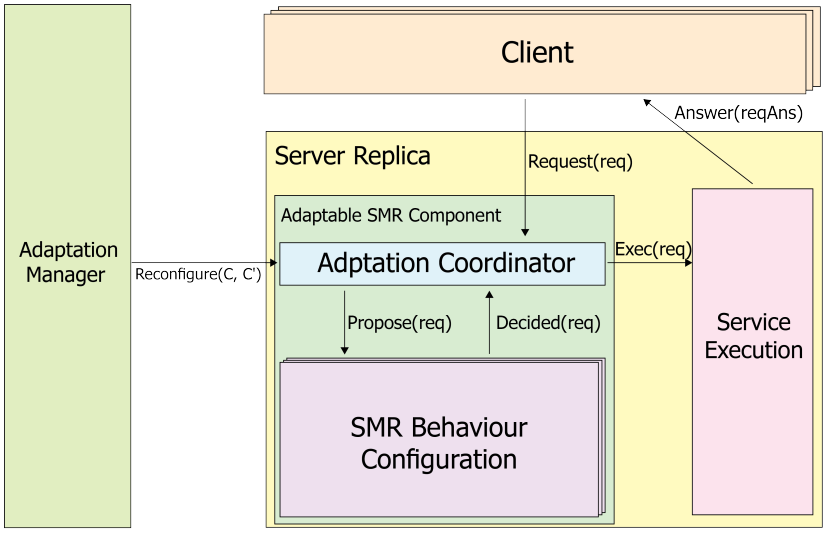

Fig. 4. The architecture of the adaptable SMR module to be implemented

The adaptable SMR component will encapsulate the state machine behaviour as an interchangeable algorithm, having an adaptation coordinator to manage this said behaviour on-the-fly.

7.1 The adaptation coordinator

The adaptation coordinator is responsible for processing the requests for adaptation that come from outside the system, run synchronization on adaptations among system replicas, if necessary, and, finally, to perform such adaptations. The coordinator will be act as a proxy both for incoming requests from clients to the system and for the delivery of the decided values by the agreement algorithm to the service execution. This will allow for the coordinator to deliver the requests to the intended agreement algorithms and to control which decided requests are delivered to execution, as they may be delivered by two different algorithms. In short, this mediation of communication will allow the coordinator to infer the state of the current configuration, so that the adaptation can be performed safely. This module can perform the switch in different ways, the most immediate and general approach is to fully stop the current algorithm and

then boot up the new one, but this would introduce a harsh loss of performance during the switch. To minimize this, it is possible to have both algorithms running simultaneously and stop the old one only when the new is fully functional, like in the works described in [18] or [19]. Yet another possible implementation is to make a three step change, like described in [20], or even build fine tuned coordination between two specific protocols [21]. So, several approaches on the implementation will be carried out, in order to compare and rationalize about them in practise. The reconfiguration module already present in BFT-SMaRt is going to be incorporated in the coordinator, so the adaptations that are already possible to be made (e.g. changing the replica set) stay available under the new architecture.

7.2 The Client

The client will also need to be aware of the adaptation, as its own behaviour is dependent of the implementation of the state machine, like for example Zyzzyva relies on clients to detect misbehaviour of the state machine replicas. As well, the pattern of communication and the decision of stable answers is very different between crash fault tolerant and BFT systems. A representation of the client's modules is depicted in Figure 5. When a client tries to send a request to the state machine using an old behaviour, that is no longer in use, the interface of the adaptable SMR will inform that the request must be made using the new protocol. To avoid that a faulty replica can be lying to the client, always refusing its requests, the client shall contact all the replicas in the system to be informed of the system configuration upon a refused request. Nevertheless, when changing between two algorithms with equivalent request messages, this is, that can be translated from one to the other, the adaptation coordinator can do this, instead of aborting the client request.

A threat to the liveness of the system may arise because a given client can be always using an old protocol, if the state machine is switching it very often. To mitigate this, the adaptation coordinator must limit the frequency of adaptations, taking into account the delays in the network, to ensure that under partial synchrony, all clients will eventually may be using the same protocol as the state machine replicas. However, if the request messages are translatable between the different SMR implementations this problem is also mitigated, as the coordinator is able to translate it, so a request made to an inactive behaviour can be translated to the currently active and processed by it.

7.3 Some strategies supported by the architecture

In this subsection we present a brief description of how some strategies can be carried out in our architecture:

– **Stop and go with stoppable algorithms:** Using a stoppable algorithm, performing a "stop and go" approach is straightforward. Upon receiving a RECONFIGURE(C, C'), the coordinator sends a stop-sign to the state machine

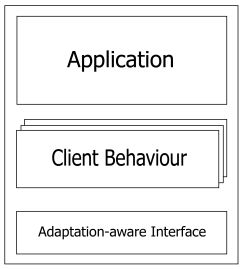

Fig. 5. The architecture of the adaptable client module to be implemented

C. Upon receiving the information that the operating machine agreed on stopping, all further commands would be addressed to the state machine implementation C'. Upon making requests, the clients are informed about the change in the configuration, so they start processing answers in the context of C', behaving accordingly to the algorithm pattern.

- **Stop and go without stoppable algorithms:** Without a stoppable algorithm, further care must be taken to decide when its safe to switch from on configuration to another. The most immediate way is to send a marked message upon the reception of a RECONFIGURE(C, C'), so that message is the last message decided by configuration C and the subsequent ones are responsibility of configuration C'. Messages decided by C after the marked message are simply ignored. Of course, if multiple agreements run in parallel, we must only shift the ordering of messages between configurations after some more messages are ordered after the marked one, as discussed in 5.3.1.
- **Speculative execution of new configurations:** One could also implement a solution similar to the one described by Bortnikov et al. Upon receiving RECONFIGURE(C, C'), the state-machine C' would be promptly started and a message stating this would be broadcast to all other replicas (through the coordinators) and a special marker would be sent for ordering in C. Upon having received a quorum, this is, in a BFT paradigm, usually $2f + 1$ messages, from other replicas stating that they are ready to run the new configuration, a replica can start ordering commands under the new configuration C'. Note that this commands are posterior to the reconfiguration one. Finally, when C decides the special configuration command, no further commands are sent to it, and then all the speculatively decided by C' commands can be executed on the execution service, after the ones decided by C.

7.4 Limitations of this solution

Some BFT SMR approaches are not fully supported by this architecture. As an example, Aardvark's replicas use independent NICs for clients and for every other replica. To support this, the communication layer of BFT-SMaRt would need to be replaced.

The problem of overloading the network when executing several protocols concurrently is not addressed by this solution. Nevertheless, we think that is possible to implement some network load conservative strategies with the present solution.

8 Evaluation

8.1 Performance of the adaptation

The performance of the adaptations will be evaluated by measuring different metrics during the switching time, using different approaches on how te adaptation is carried out. The first one is going to be the time of unavailability to answer to requests (if there is a period where no algorithm is making progress), as we aim to have a system capable of adaptations that take little to no downtime. The throughput is another key performance metric, as even if there is no downtime a significant drop in throughput may not be tolerable in some systems. The bandwidth consumed by the adaptation coordination is also an important factor to take into account, as it may be a limiting factor, if the SMR implementation operates near the maximum bandwidth available.

This measurements will be taken both when the system is running under favourable conditions and under more unstable conditions. This is, both when the network is stable and there are no failures in the replicas, and when there is starvation in the network, or packets are being lost often or even when there are failing replicas. We think evaluating the system in both situations is relevant, as the first reflects a use case when the adaptation is performed preemptively, and the latter reflects a case of an adaptation that is taken when a system is already suffering of performance issues.

8.2 Usefulness of the adaptive system

We will measure the usefulness of this work by comparing the behaviour of an adaptive system built from our solution with a static SMR implementation. The tests will be made under a stable workload, workload changes and even other environmental changes, as an increase in the latency of the network (which happens often in wide-area networks) or the presence of malicious Byzantine clients.

We expect that under a stable workload our system will be slightly less performant than a fitting static SMR implementation is chosen. Although we aim to minimize it, the adaptability of the system will always carry some overhead due the extra processing and messaging necessary to support such adaptations.

We hope that under workload and environmental changes, our system will surpass the performance of some static approach. To measure this, we will measure the throughput of requests answered under a given workload and given adaptations. As this system has no capability to decide when and what adaptations to perform, we will mock a system that will provide such suggestions of

change. We will make an effort so the workload changes and the adaptations performed depict realistic use cases and provide meaningful data to rationalize about. The environmental changes will be injected during the tests, developing clients that will try to render the system slow or even unusable, as well as components in the network to introduce delay and instability, trying to simulate wide-area or lossy networks.

9 Scheduling of Future Work

Future work is scheduled as follows:

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

10 Conclusions

The use of Byzantine fault tolerant systems is becoming more and more relevant, not only because computer systems tend to deal with highly critical data and decision, but also because the deliberate attacks and arbitrary faults are becoming more prominent. To deal with this, a lot of BFT solutions were developed, each one trying to surpass the performance and resiliency of its pairs. Although, the optimizations done always carry some kind of trade-off, making hard to develop a system that is best for all kinds of usages and environments.

To leverage the advantages of all the BFT solutions already developed we propose a BFT SMR system that is capable of switching between different protocols on-the-fly. This way, at any given point in time, is possible to have the best BFT protocol running for the current environment (workload, existing failures, network conditions, etc.), without the need for a human administrator to manually shut-down the system and reboot under a new configuration.

We have presented a view of the architecture of the proposed solution, as well as how we intend to test and validate it. Its detailed specification, implementation and evaluation will be done as future work, accordingly to the schedule already presented.

Acknowledgments

We are grateful to Daniel Porto, Manuel Bravo, Miguel Pasadinhas, Bernardo Palma and Pedro Joaquim for the fruitful discussions and comments during the preparation of this report. We are also grateful to Adelaide and Daniel

André, Diana Almeida and António Venâncio for the discussions about the meta-approaches to this work. This work has been partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects with references PTDC/ EEI-SCR/ 1741/ 2014 (Abyss) and UID/ CEC/ 50021/ 2013.

References

1. Lamport, L.: Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research (2005)
2. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3) (July 1982) 382–401
3. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. OSDI '99 (February 1999) 173–186
4. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07, ACM (October 2007) 45–58
5. Singh, A., Das, T., Maniatis, P., Druschel, P., Roscoe, T.: Bft protocols under fire. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation. Volume 8 of NSDI'08., USENIX Association (2008) 189–204
6. Bessani, A., Sousa, J., Alchieri, E.E.: State machine replication for the masses with bft-smart. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, USA, IEEE (June 2014) 355–362
7. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) **22**(4) (December 1990) 299–319
8. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), Philadelphia, PA, USA, USENIX Association (June 2014) 305–319
9. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4) (December 2001) 18–25
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2) (April 1985) 374–382
11. Copeland, C., Zhong, H.: Tangaroa: a byzantine fault tolerant raft
12. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2) (April 1980) 228–234
13. Diffie, W., Hellman, M.: New directions in cryptography. IEEE transactions on Information Theory **22**(6) (November 1976) 644–654
14. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. NSDI'09, Berkeley, CA, USA, USENIX Association (April 2009) 153–168
15. Aublin, P.L., Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 bft protocols. ACM Transactions on Computer Systems (TOCS) **32**(4) (January 2015) 12
16. Rosa, L., Rodrigues, L., Lopes, A.: A framework to support multiple reconfiguration strategies. In: Proceedings of the Autonomics 2007, Rome, Italy (October 2007) 15

17. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. ACM SIGACT News **41**(1) (March 2010) 63–73
18. Mocito, J., Rodrigues, L.: Run-time switching between total order algorithms. In: Proceedings of the Euro-Par 2006. LNCS, Dresden, Germany, Springer-Verlag (August 2006) 582–591
19. Bortnikov, V., Chockler, G., Perelman, D., Roytman, A., Shachor, S., Shnayderman, I.: Reconfigurable state machine replication from non-reconfigurable building blocks. arXiv preprint arXiv:1512.08943 (2015)
20. Chen, W.K., Hiltunen, M.A., Schlichting, R.D.: Constructing adaptive software in distributed systems. In: Proceedings of the 21st International Conference on Distributed Computing Systems, IEEE (April 2001) 635–643
21. Couceiro, M., Ruivo, P., Romano, P., Rodrigues, L.: Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. IEEE Trans. Parallel Distrib. Syst. **26**(11) (November 2015) 2942–2955
22. Lamport, L., Malkhi, D., Zhou, L.: Stoppable paxos. TechReport, Microsoft Research (2008)
23. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (October 2009) 277–290
24. Lorünser, T., Happe, A., Slamanig, D.: Archistar - a framework for secure distributed storage. http://ARCHISTAR.at (2014) GNU General Public License.
25. Sousa, J., Bessani, A.: From byzantine consensus to bft state machine replication: A latency-optimal transformation. In: Ninth European Dependable Computing Conference (EDCC), 2012, IEEE (May 2012) 37–48
26. Sabino, F., Porto, D., Rodrigues, L.: Bytam: um gestor de adaptação tolerante a falhas bizantinas. In: Actas do oitavo Simpósio de Informática (Inforum), Lisboa, Portugal (September 2016)