Adaptive BFT protocols

Frederico Sabino
frederico.sabino@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Byzantine fault-tolerant (BFT) protocols have been proposed mostly as a means to support the execution of highly-resilient services, i.e., of services that keep their availability, integrity, and confidentiality even if there are accidental or malicious faults (attacks, intrusions). Several distinct BFT protocols have been proposed in the literature and each of these protocols excels for different operational conditions. Furthermore, even for the same protocol, the configuration and deployment options (number of replicas, characteristics of servers, etc) have a huge impact on the performance of the system. Since the conditions that affect the performance of a BFT protocol, such as the workloads and the threats, may change in runtime, it is relevant to devise techniques to adapt BFT implementations and/or deployments dynamically. This work proposes a number of viable adaptations of BFT protocols, for a number of scenarios that are meaningful in practice, and studies the implementation of BFT protocols that can support these adaptation in runtime. In this context we also study what are the relevant parameters that need to be monitored to trigger the adaptations and what are the policies that can be used to drive the adaptations in a secure manner.

1 Introduction

One of the most relevant techniques to implement distributed dependable systems is the State Machine Replication (SMR) [1]. This is a general approach that can be applied to all services that can be implemented by a deterministic state machine: it consists in running multiple instances of the service in different machines, with independent failure modes. Even if one or more machines fail, the remaining machines can continue to provide the service (given some ratio among faulty and correct machines that depends on the type of faults that need to be tolerated). In order to keep the correct replicas consistent, SMR requires commands to the state machine to be disseminated using an *Atomic Broadcast* Protocol [**?**] that ensures that all replicas receive the same set of commands exactly in the same order.

Atomic Broadcast is an instance of *Consensus*, a fundamental problem in distributed systems that has been extensively studied in the literature [**?**]. Many different consensus protocols have been derived for different system models (for instance, for synchronous and asynchronous systems) and for different fault models (for instance crash faults, arbitrary faults, etc). In our work we are particularly concerned with protocols that can tolerate arbitrary faults because these

protocols are resilient not only to faults caused by natural phenomena (bit flips, electric sparks, etc) but also to faults caused by malicious adversaries, such as an intruder that takes control over a given machine. Protocols that tolerate this type of faults are also known as Byzantine fault-tolerant (BFT) protocols, after a famous paper by Lamport *et al.* [2].

The increasing risk of facing arbitrary faults caused by malicious attacks on systems, boosted the research on Byzantine fault-tolerant protocols, specially after the work by Castro *et al.* [3], that demonstrated that it is possible to build BFT services with acceptable overhead. Several distinct BFT protocols have been proposed in the literature and each one of these protocols excels for different operational conditions [4–7]. Furthermore, even for the same protocol, the configuration and deployment options (number of replicas, characteristics of servers, etc), have a huge impact on the performance of the system. Since the conditions that affect the performance of a BFT protocol (such as workloads and threats) may change in runtime, it is relevant to devise techniques to adapt BFT implementations and/or deployments dynamically.

In this work we study the design and implementation of adaptive Byzantine fault-tolerant systems, i.e., systems that execute BFT protocols and are able to adapt the protocol, its configuration, or the deployment choices in response to changes in the operational envelope. An aspect of critical importance for this work is to derive implementations of BFT protocols that are adaptable, i.e., that can change their configuration, parameters, and even algorithms in runtime in a secure manner. This involves using modular designs in the implementation of the protocols, such that it is possible to change only a subset of the protocol behaviors if needed. Also, it requires the use of a robust adaptation manager that is capable of selecting when and how to adapt based on information collected from the environment and on a set of policies that capture the high level goals that the system must satisfy.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Sections 3, 4 and 5 we present all the background related with our work. Section 6 describes the proposed architecture to be implemented and Section 7 describes how we plan to evaluate our results. Finally, Section 8 presents the schedule of future work and Section 9 concludes the report.

2 Goals

This work addresses the problem of implementing and evaluating adaptive Byzantine fault-tolerant systems. In particular we aim at:

> *Goals:* Assessing the benefits and limitations of using dynamic adaptation to increase the throughput of systems that rely on Byzantine fault-tolerance without decreasing their resilience.

To achieve this goal we will identify a set of viable adaptations of BFT protocols, for a number of scenarios that are meaningful in practice. We also identify

what are the relevant parameters that need to be monitored to trigger the adaptations and what are the policies that can be used to drive the adaptations in a secure manner. Finally, to implement the adaptations we plan to implement an adaptable version of the BFT-SMaRt [8] implementation, by augmenting it with additional algorithms and the mechanisms required to commute in runtime among these algorithms.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the architecture needed in order to change between BFT protocols; ii) an implementation of an extension to the BFT-SMaRt supporting protocol interchangeability iii) an extensive experimental evaluation on the system adaption using different stress scenarios.

3 Byzantine Fault Tolerance

In this section we make an overview of the work on Byzantine Fault Tolerance. In Section 3.1, we start by discussing how Byzantine faults relate to other types of faults typically considered in fault-tolerant systems. Then, in Section 3.2, we describe the model that characterizes the system we are targeting. We proceed to enumerate the properties a BFT protocol must preserve in Section 3.3. In Section 3.4 we describe some of the most relevant BFT protocols that have been proposed in the literature and in Section 4 we survey techniques that have been proposed to reconfigure BFT protocols. Finally, in Section 5, we describe the BFT implementation that we plan to use as the basis for our development work.

3.1 Fault Model

We consider a distributed system composed of n nodes, out of which f may be faulty. The relationship between n and f is usually called *resilience* [9] and depends on the type of faults that one aims at tolerating (and also on the system model, described in the next section). Not surprisingly, some faults are easier to tolerate than others. A possible classification of faults, in increasing order of severity, is captured by the following list of fault types:

- *Crash-stop*: a crash fault occurs when a process stops executing requests/steps. A process that crashes executes the algorithm correctly until some point in time t, when it stops operating a never recovers. Note that this model does not prevent a crashed machine from recovering, however it is assumed that after recovery, processes need to join the system as new processes.
- *Omission*: an omission fault occurs when a process does not send or receive a message that was supposed to be sent or received according to the algorithm. Omission faults are more general than crash faults as the faulty process may continue its execution after an omission occurs (unlike crash faults, where it is assumed that the process stops after the fault).

- *Crash with Recovery*: in this mode, a process is considered faulty if it either crashes and never recovers or is constantly crashing and recovering. A process that crashes and recovers a finite number of times during an execution is still considered to be correct. When a process recovers, after a crash, it may need to update its internal state before interaction with other processes. Thus, techniques that make usage of a stable storage and logging mechanisms are deployed to help the recovery procedure. To ease the need of stable storage we might consider that a set of processes never crash, so the requirement of having a stable storage is no longer required.
- *Eavesdropping:* this kind of faults may occur when an active adversary is present on the system. Leaks of private information exchanged by processes fall in this fault category. A faulty process may leak information about its internal state and possibly give the attacker enough information to coordinate an attack. These faults can be prevented by applying secure cryptographic techniques to the private data.
- *Arbitrary*: faults that cause arbitrary deviations from the correct algorithmic behavior fall under this class. By making no restriction on what can happen when a fault occurs, the arbitrary fault model is the most general. Arbitrary faults may have natural causes (for instance, electromagnetic interference), result from unintentional errors (a bug in a program), or be intentionally generated by a malicious attacker that was able to exploit a vulnerability of the application or of the operating system. Arbitrary faults are also often named Byzantine faults [2, 9].

From the faults hereby described, the focus of this work is on systems that are able to tolerate arbitrary faults.

3.2 System Model

The types of faults are not the only factor that affect the resilience of the system. Other characteristics of the system, namely its behavior in the time domain, are also relevant. In fact, even relatively benign faults such as crash-stop may be hard to detect (and to tolerate) in a distributed system unless assumptions can be made regarding communication and processing delays in the system. In this context, it is important to distinguish the following relevant models of distributed systems: synchronous, asynchronous, and partially-synchronous systems.

- *Synchronous System*: a system is considered synchronous if there is a known upper bound on the processing delay (synchronous computation) and a known upper bound on the message transmission (synchronous communication). Since there is a known upper bound established on the delays of the system, it becomes much easier to detect failures: if a given process does not respond timely, it is considered to be faulty. However building such a system requires an overall analysis of the complete infrastructure that composes the system such as hardware and software limitations; only then it is possible to conclude what might be a feasible upper bound given the current limitations.

- *Asynchronous System*: a system is considered asynchronous if there is no timing assumptions on the processes and network links. Thus, in an asynchronous system we have no upper time bounds for communication and processing delays.
- *Partially-Synchronous System*: in practice, systems are designed and deployed such that they can respect pre-defined time bounds in normal operational conditions, i.e., they behave like synchronous systems most of the time. However, they are also not over-provisioned for the worst case and, therefore, may experience long delays under stress, for instance when the network or the CPU is overloaded due to a peak in the workload, i.e, they may behave transiently as an asynchronous system. These system can be modeled by the property of *eventual synchrony*: a property that states that the system will eventually behave as a synchronous system although there is no certainty of when this is going to happen. However it is not assumed that, when reaching synchrony, the system will stay synchronous or that its initial state is in an asynchronous period [9]. Many practical solutions for addressing Byzantine fault tolerance assume this model [4, 7, 9, 10].

3.3 BFT Concepts and Primitives

In this report we are interested in studying protocols that can help to implement replicated services that can tolerate Byzantine faults. As most of the literature on the subject, we will consider services that can be implemented using *state machine replication* (SMR). The SMR approach, originally proposed by Lamport[1, 2, 11] and adopted by most BFT approaches[3–7] assumes that the service can be modelled as a state machine. The server starts with some initial state S_0 and then evolves by executing a sequence of commands c_1, c_2, ..., making a deterministic state transition in each step (i.e., the next state depends exclusively of the previous state and of the command being executed). As a result, if two correct replicas start with the same initial state, and execute the exact same sequence of commands, they end up in the same final state. Clients send commands to all replicas and receive back replies, and then apply some function to the set of replies received to mask server failures. If only crash failures can occur, $f + 1$ replicas are needed and the client can simply pick the first reply. If Byzantine failures can occur, at least $2f + 1$ replicas are needed, such that the client can always get a majority of correct replies (and discard the faulty replies, if any).

As described above, to ensure that correct replicas keep a consistent state, one needs to ensure that all correct replicas receive the same set of commands in the same order. Therefore, a protocol is needed to ensure that all commands are reliably broadcast to all replicas and totally ordered among each other. A protocol that achieves such goal is called an *atomic broadcast* protocol[9] (the designation "atomic" derives from the fact that it remains indivisible despite failures, the message is either delivered to all correct replicas or to no correct replica and delivery appears to be "instantaneous" in time, resulting in a total

order of all deliveries). Atomic broadcast can be implemented as a serial execution of several consensus instances where, in each consensus instance, one command is decided, i.e., replicas run consensus instance number one to decide the first command to execute, then they run consensus instance number two to decide the second command to execute, and so forth.

3.3.1 Byzantine Consensus We now provide a more precise description of the properties that Byzantine Consensus satisfies. Distributed protocols can be characterized by safety and liveness properties, as described below:

- Safety: a safety property states that bad things do not happen[9]. In the consensus problem, a set of processes propose values and they have to decide on the on the same value. A safety property for consensus should state that the decided value is the same for all the processes and that the decided value was proposed by some correct process p.
- Liveness: a liveness property states that good events will eventually happen [9]. A liveness property for the consensus problem should state that the all processes will eventually decide on a value.

More precisely, Byzantine Consensus can be characterized by the following properties[9]:

- *Termination:* Every correct process eventually decides some value.
- *Strong validity:* If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or a special value $\perp$ (which means that no command is executed).
- *Integrity:* No correct process decides twice.
- *Agreement:* No two correct processes decide differently.

When used to implement atomic brodcast, the value v is a command issued by a client.

3.3.2 Views As it will become clearer in the subsequent description, most protocols that implement Byzantine Consensus use a leader-based approach. Generally speaking, one of the replicas is elected as a leader and acts as a coordinator to facilitate the agreement process. If the leader is non-faulty, and is not suspected to be failed by the other non-faulty processes, it selects a client command and coordinates with other processes to ensure that such command is decided. If the leader stops, becomes extremely slow, or is Byzantine and does not comply with the specified protocol in a way that can prevent consensus from being reached, the remaining correct processess initiate a procedure to replace the leader, a process that is called a *view change*. A view is, therefore, a list of processes that are participating in the consensus instances and their roles (which one is the leader). Since to implement atomic broadcast multiple instances of consensus need to be executed (one for each command), the same view is used in consecutive instances, until a reconfiguration is needed.

3.3.3 Recovery and Membership Changes

In a system that implements state machine replication, it is usually helpful to be able to add new replicas or to allow replicas that have stopped (either due to a crash or due to schedule maintenance) to later re-join the system. We have already discussed that most Byzantine SMR systems suppor a view change operation. This operation can be used not only to change the roles of the participating processes (i.e, to select a new leader) but also to add or remove replicas from the system. However, adding replicas to a running system introduces the additional problem of bring the state of the replica up-to-date. This can be achieved by letting the new replica execute all the commands that have been executed by the active replicas, by copying the state form a correct replica, or by a combination of these approaches. For this purpose, important extensions like logging and checkpoint creation, state transfer and reconfiguration, greatly influence the protocol's execution.

- State Transfer: adding or removing replicas to a system can contribute to the extension of its lifetime while continuing to process clients' requests. The ideas of state transfer and reconfiguration are further explored in Section 4.3.1.
- Logging and Checkpoint Creation: logging is a technique used to store the current actions performed by a given replica. When a reconfiguration to the system is being performed some replicas may be in transient state trying to recover the current state. Logging the actions performed help the replica to reach that state. However, the log can grow to sizes that become computationally exhaustive to store or to perform a recovery on. Thus we can save a "snapshot" of the current process state to a reliable storage so that it can recover based on that information stored – this is called a checkpoint and it discards the log entries made until the taking of the "snapshot" thus reducing greatly the space requirements by the system. A recovering replica now only needs to ask for the last checkpoint in the system and build up the log from there.

The work [12] demonstrated that consensus can indeed be solved under a system that is partially-synchronous which represents the class of systems that this report focuses on.

3.4 BFT Protocols

One of the main barriers while deploying a BFT protocol is the high requirements needed to tolerate f faults – usually the number of servers n, must be greater or equal than $3f + 1$.

With different protocols being more adequate for different workloads and environment conditions a system designer would need to choose the right technique under the assumption that while the system is running, the conditions will not deviate from the expected.

In this section we describe the Byzantine Generals Problem [2] which formalizes the problem of arbitrary faults. Then we describe three works that offer

a solution to the problem described: PBFT [3], Zyzzyva [4] and Aardvark [7]. PBFT was the first practical system that tolerated byzantine faults under an asynchronous environment while Zyzzyva and Aardvark further explored variations of that system. Other variations such as HQ [6] and Q/U [5] are not discussed. We chose Aardvark and Zyzzyva because they represent two protocols in completely opposites sides of the spectrum: Aardvark is more robust so it does not deviate much from its usual communication pattern when in presence of faults while, on the other end of the spectrum we have Zyzzyva, a solution that offers high client throughput but is more fragile, i.e., in the presence of faults, it executes a fallback mechanism with an additional "recovery" step. These protocols should give insight of what the current state of the art is regarding solutions to the Byzantine problem under SMR.

3.4.1 The Byzantine Generals Problem Before beginning with the description of the protocols used to tolerate arbitrary faults, it is important to understand the underlying problem that those protocols try to solve. The description of the problem was formalized by Lamport in [2].

In 1978, Lamport L. depicted an algorithm that introduced the concept of SMR under a distributed environment [1]. From the concept of partial ordering, the algorithm described an extension that provided total ordering – the algorithm however produced arbitrary results if it was not coherent with the decision of the system's users. A solution for this arbitrary problem is by using synchronized clocks.

Additionally, the algorithm assumed that processors never failed and that all the messages were correctly delivered – it operated on a non-faulty system. Lamport then created a real-time algorithm [11] that assumed upper bounds on message delays and that correct processes had their clocks synchronized. It is one of the first algorithms that described the idea of arbitrary faults described in The Byzantine Generals Problem [2].

The scenario used for explaining the Byzantine Generals Problem is an attack performed to an enemy city by the Byzantine army which has divisions commanded by their own general. Communication between generals are only possible via messenger. The objective of the Byzantine army is to reach a plan. However some generals are traitors and may disrupt consensus from happening.

From this point onwards, and in order to directly compare to other solutions, we use the term *primary* to describe a general and a the term *backup* to describe a lieutenant and a traitorous behavior means that the instance (primary or backup) is faulty.

The algorithm used by the instances must have the following guarantees:

- All correct instances decide upon the same plan of action
- A small number of faulty instances cannot cause the correct instances to adopt a bad plan

Every primary communicates its information to other instances – $v(i)$ is the information communicated by the ith primary and by combining $v(1), ..., v(n)$

a primary can reach a plan of action. However, this solution does not work – a faulty primary may send different values to different instances which means that correct instances will have conflicting values. Therefore, two new requirements must be formalized:

- Any two correct instances use the same value of $v(i)$
- If the ith instance is loyal, then the value that he sends must be used by every correct instance as the value of $v(i)$

Since the new requirements focus on the the value sent by one instance (the primary), the problem can be reduced: given an army of n instances, there is a single primary instance which sends orders to his $n-1$ backup instances. The problem is now reduced to two *interactive consistency* conditions:

- **IC1**: All correct backup instances obey the same order.
- **IC2**: If the primary is correct, then every correct backup obeys the order he sends.

We can also conclude that if the primary is correct then $IC2 \Rightarrow IC1$. The original problem can be viewed as an application of these conditions: each instance can be seen as a primary which sends his order $v(i)$ and all other instances act as backup instances.

Now let's suppose we have a primary and two backup instances (Backup 1 and Backup 2) and two possible decisions: attack or retreat. Additionally one of the three instances is faulty.

If Backup 2 is faulty, and the primary issues an attack order to Backup 1 and 2 then Backup 2 may also report to Backup 1 that the order given was to retreat. To satisfy IC2, Backup 1 should proceed with the attack order.

If the primary is faulty he may issue two different orders to Backup 1 and 2 respectively – by sending an attack order to Backup 1 and a retreat order to Backup 2, Backup 1 must obey the attack order since he does not know which instance is the faulty one. Similarly, Backup 2 will follow the retreat order and violated IC1.

Concluding, no solution exists for three instances in the presence of one fault. From the result of [13] we need at least $3f+1$ instances to tolerate f faults.

Oral Messages Solution: A first solution with $3f+1$ instances (where we have f faults) used what is called oral messages with the following assumptions:

1. Every message that is send is delivered correctly.
2. The receiver of a message knows who sent it.
3. The absence of a message can be detected.

Additionally the primary sends messages to all other backup instances. If the primary is faulty and decides not to send any message then the default action of the backups that did not receive the message is to retreat.

The Oral Messages algorithm $OM(f)$, copes with f faulty instances and assumes a *majority* function. The steps are described in [2] but a quick summary of the algorithm is as follows:

1. The primary sends a command to every backup.
2. Each backup acts as a primary and sends out a command to all other backups.
3. Use the majority function to compute the value based on the commands received in step 2.

A recursion of the algorithm occurs in step 2 – each backup acting as a primary will start the algorithm with $OM(f-1)$. So for $OM(f-k)$ the algorithm will be called $OM(n-1)...OM(n-k)$ times.

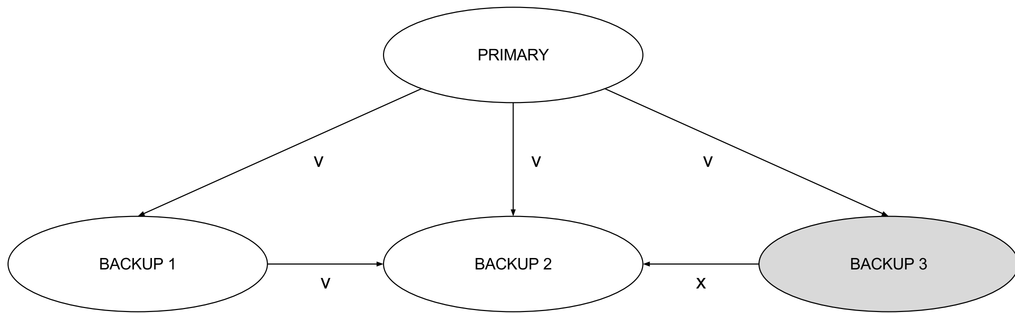

Fig. 1. Backup 3 is faulty

In Figure 1 we have the case where $f = 1$ and a backup is faulty. We can see that both Backup 1 and 2 receive the values $\{v, v, x\}$ therefore IC1 is respected because the majority of the set of responses is v. IC2 is also respected because all loyal backup instances (1 and 2) respect the correct primary's order.

In Figure 2 we have the case where $f = 1$ and the primary is faulty. Despite the primary sending different orders to its backup instances, all backups arrive to the same set of orders $\{x, y, z\}$ so IC1 is respected (IC2 is not relevant since the primary is faulty).

As a remark, this is a simple example where we have only one fault. Remember that the algorithm is recursive so the communication is quite expensive in each step.

Signed Messages Solution: In the Oral Messages solution, a backup could lie about the primary's order. If that ability is restricted the problem becomes easier – by signing messaged, we can add two additional assumptions to the messages interchanged between instances:

1. A correct primary's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.
2. Anyone can verify the authenticity of a primary's signature.

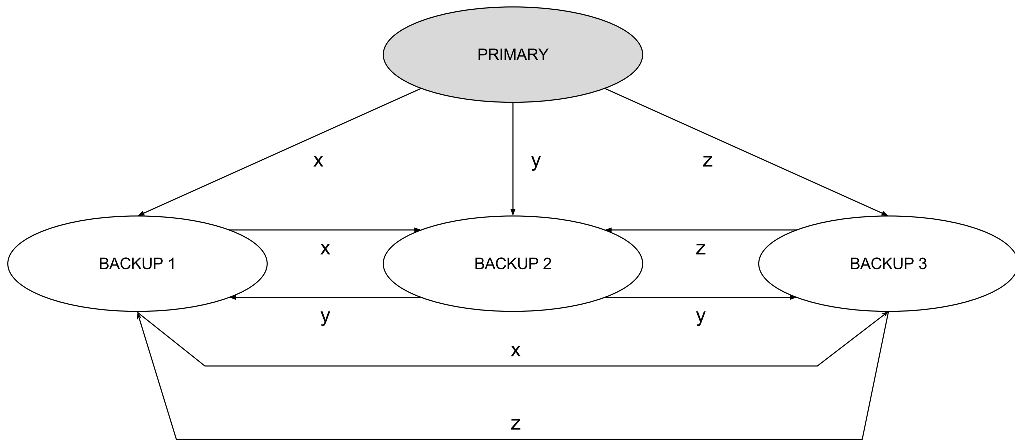

Fig. 2. Primary is faulty

The Signed Messages algorithm, $SM(f)$, proposed in [2] tolerates f faults but the requirement of at least 3 instances no longer holds – the problem can be solved with 3 instances given $2f + 1$ for the number of total instances. In this algorithm, each backup maintains a set V that stores received orders that were properly signed. A new function, $choice(V)$ is also assumed by the algorithm and it needs to have the following requirements:

1. $choice(V) = v$ if V contains only one element
2. if V is an empty set then $choice(V) = RETREAT$

The algorithm starts by the primary sending a signed order to his backup instances. A backup when receives an order (either by a primary or other backups) verifies its signature and, if valid, puts it in V. If the backup has received k signed messages and $k < f$ then he sends v to all other backups that might have not seen v. When Backup i does not receive any more messages, he obeys the order given by $choice(V_i)$.

All loyal backups will eventually compute the same set V and by using a deterministic function $choice(V)$, IC1 is respected. Also if the primary is loyal IC2 is also respected (all correct backup instances will have the same V).

In Figure 3 is a demonstration of an execution of $SM(1)$ where the primary is faulty. The message *"attack":0* means that the order *attack* was signed by the instance 0 (primary). Additionally *"attack":0:1* means that the previous message was additionally signed by Backup 1. In this situation, the primary issued an “attack” order to Backup 1 and a “retreat” order to Backup 2. After the communication step, each Backup arrived to the same set where $V_1 = V_2 = \{\text{"}attack\text{"}, \text{"}retreat\text{"}\}$. Each Backup arrives to the conclusion that the primary is faulty since he signed two different orders and the signature cannot be forged.

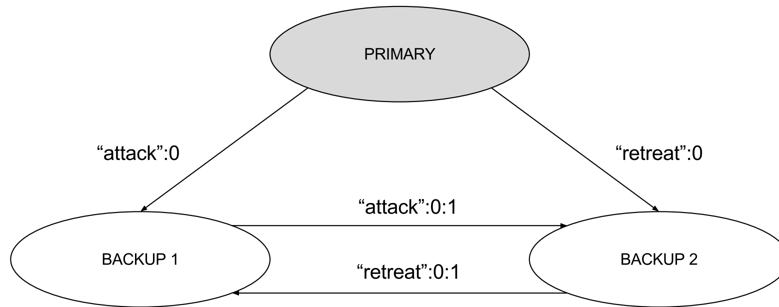

Fig. 3. Primary is faulty

An important remark must be made about the assumptions of the messages – a faulty instance can delay or not transmit at all a given message but it can be detected. To detect this failure there are two timing assumptions being made (which are dominant in synchronous systems):

1. There is a fixed maximum time needed for the generation and transmission of a message.
2. The sender and receiver have clocks that are synchronized to within some fixed maximum error.

Despite the formalization of two solutions in this section (one requiring $3f+1$ replicas and the other requiring $2f+1$ replicas), these solutions were designed for synchronous systems so we cannot assume that they work on an asynchronous environment like the Internet.

PBFT and variants explained in this report (Sections 3.4.2, 3.4.3, 3.4.4) are based on SMR and require $3f+1$ replicas. To expand on the requirement for the number of replicas needed in an asynchronous system, there are two relevant properties that need to be stated:

- *Intersection*: any two quorums have at least one correct replica in common.
- *Availability*: there is always a quorum available with no faulty replicas.

In order to provide liveness we need to assume that we will only receive $n-f$ responses (f replicas may not respond) – this is the quorum size ($= 2f+1$). To provide correctness, two quorums must intersect at least in one correct replica: $(n-f)+(n-f)-n \geq f+1$ thus the result $n \geq 3f+1$.

The systems also assume that a strong adversary can coordinate faulty nodes in order to compromise the replicated service. However, the adversary cannot break cryptographic techniques (like collision-resistant hashes, encryption, signatures and authenticators). These systems use signed messages and/or MACs for authentication purposes (usages between these two methods are described when relevant for each protocol). They are Partially-Synchronous systems where we can assure both *liveness* and *safety* properties when we have synchrony but only

liveness is guaranteed in periods of asynchrony [14]. It is also assumed that there is a finite number of clients where any number of which can be faulty. The protocols also follow a primary-backup mechanism: within a given view, one replica is the primary (leader) and the others are backups.

3.4.2 PBFT Castro and Liskov, introduced the first BFT protocol that is safe under asynchronous systems – PBFT [3].

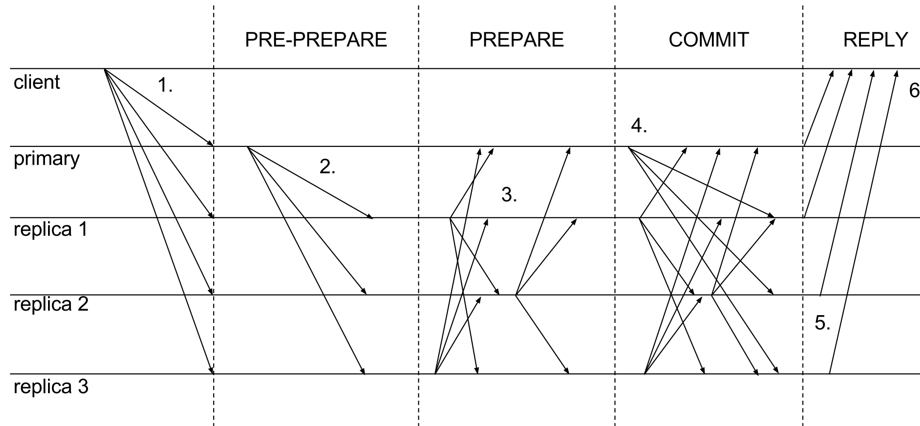

Fig. 4. PBFT Communication Pattern

PBFT uses a three-phase commit communication pattern: PRE-PREPARE, PREPARE, and COMMIT. The messages are authenticated using a MACs. PRE-PREPARE and PREPARE are phases that totally order the requests for a given view (even when the primary, which is responsible for the ordering, is faulty). Each replica keeps a message log to register the messages that it has received or sent.

1. Client sends request to a replica – a client sends a request to what it believes to be the primary. If it doesn't receive a response within a time bound then it retransmits the request to all other replicas. The replica then verifies the message by checking if the client's request is within the load limits of the server. If a replica, other than the primary, receives the client request, it authenticates the message and sends it to the primary.
2. Primary sends a PRE-PREPARE message to all Replicas – When the primary server receives a client request, it assigns a sequence number n and registers the message in its log. Then it multicasts a PRE-PREPARE message to all other backup replicas containing the message and the current view number v. A replica only accepts a message if it is in the same view and its successful in the verification of its authenticity.

3. Replica receives PRE-PREPARE from the primary and sends PREPARE to all other replicas – upon receiving the PRE-PREPARE message, the replica authenticates the message. If the replica has already accepted the message then it is discarded. If the replica has already processed the message (same sequence number) in the current view, then it is discarded. The replica also checks for the integrity of the MAC authenticator used in the message: if it is invalid, the message is discarded. If the authenticator is valid then the replica registers the PRE-PREPARE message and sends a PREPARE message to all other replicas along with a digest of the requests present in the PRE-PREPARE message.
4. Replicas then collect a quorum of $2f$ matching PREPARE responses proving that a quorum has agreed to assign the sequence number n to the message in view v. However, this is not sufficient to guarantee that each correct replica has the same sequence of messages in the same view as another correct replica – a replica might have received the prepare message in a different view but the request has the same sequence number. That is why there is a commit phase – each replica multicasts a COMMIT message stating that it has indeed received a quorum of prepared messages (the commit message is added to the log). When a replica receives $2f$ commit messages it already has the needed quorum since the replica itself is prepared to commit the request. The replica only executes the request after executing any pending requests with lower sequence numbers.
5. Replica receives $2f+1$ COMMIT messages and sends a REPLY to the client – after receiving $2f+1$ matching responses (from distinct replicas), the replica executes the request, and sends a REPLY to the client containing the result of the execution.
6. The client then waits to receive $f+1$ messages with valid MACs and with the same result and time in which the request has been made and concludes the execution.

3.4.3 Zyzzyva

Zyzzyva is a BFT-protocol that uses speculation in order to reduce the cost of the BFT replication. In this protocol, replicas reply to a client's request without running an expensive consensus protocol. Instead, replicas rely only on the order proposed by the primary server, then process the ordered requests and immediately respond to the client. The client however has an additional task: if the client detects any inconsistencies with the responses received, it helps in the convergence of the responses by the correct replicas. By using speculative execution, Zyzzyva can achieve high request throughputs.

The protocol relaxes the condition: *a correct server only emits replies that are stable*. Instead, realizing that this condition is stronger than needed, Zyzzyva's core idea is based on the role of the client in the system: *a correct client only acts on replies that are stable*. This weaker condition avoids the all-to-all communication between replicas to reach consensus.

Ultimately, the challenge is ensuring that responses to correct clients become stable. While this task is given to the replicas, a correct client can greatly speed

the process by supplying information that would make the request become stable or even lead to the election a new primary server.

Zyzzyva's SMR protocol is executed by $3f+1$ replicas and it is based on three subprotocols: agreement, view change and checkpoint. The *agreement* protocol executes within a sequence of views where a replica(*primary*) leads the agreement subprotocol. The *view change* subprotocol is responsible for the election of a new primary (due to the current faulty one). The *checkpoint* subprotocol reduces the state that is stored within each replica thus optimizing the *view change* subprotocol.

A request completes at a client when the client has a sufficient number of matching responses ensuring that all correct replicas will execute the request. The client can determine the when the request completes since the client receives responses from replicas that include the reply to the application and the history. The history contains all requests executed by the replica prior to (but including) the current request.

There are three cases that are considered for Zyzzyva's agreement protocol: the *fast case*, the *two-phase case* and the *view change* case.

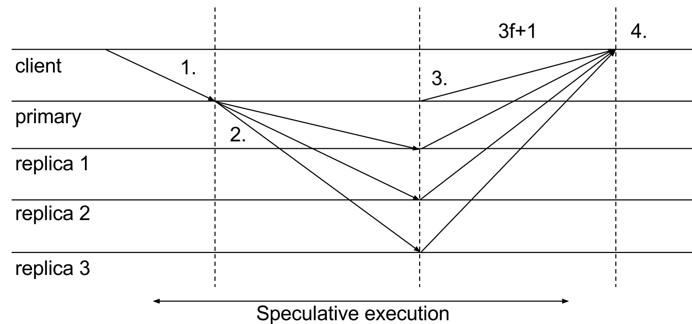

Fig. 5. Zyzzyva's *fast case* Communication Pattern

In Zyzzyva's *fast case*:

1. The client sends a request to the primary. The request sent by the client to the primary has a timestamp that guarantees only one execution of the request (exactly-once semantics).
2. Upon receiving the request, the primary assigns a sequence number and adds a cryptographic one-way hash for the request sent by the client. The primary then forwards the ordered request to all other replicas.
3. The replica receives the ordered request from the primary and optimistically assumes that the primary is correct, adds the request to its history, speculatively executes it and responds to the client. Additionally, the replica only executes the request if the message from the client is well-formed, the digest matches the cryptographic hash of the message and the primary forwarded the message in the same view where the replicas receive it.

4. The client receives $3f + 1$ matching responses and completes the request. Without any faults or timeouts the $3f + 1$ responses will match and the client can safely rely on the request result.

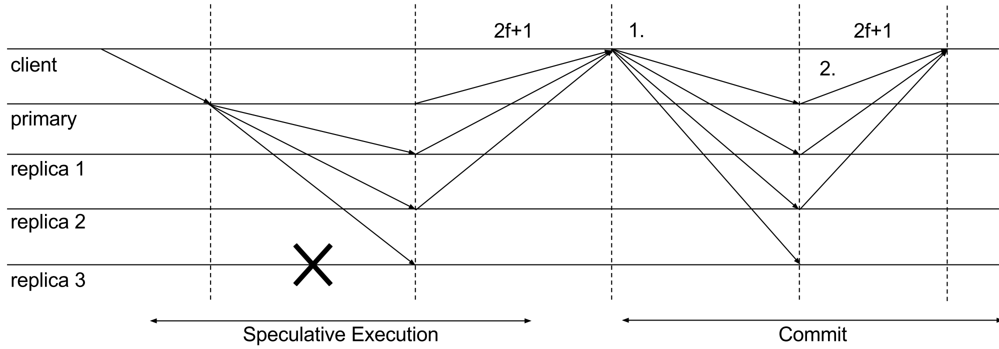

Fig. 6. Zyzzyva's *two-phase* Communication Pattern

The Zyzzyva's *two-phase* communication pattern happens if some of the replicas are faulty or slow. It applies when the client receives between $2f + 1$ and $3f$ responses. All the communication is exactly as it was described in the *fast case* except for the client's reception of the responses.

1. The client now receives between $2f+1$ and $3f$ responses, assembles a *commit certificate* and relays it to all the replicas. When the client sends the request, it sets a timer. After the timer expires, if the client gathered between $2f+1$ and $3f$ matching responses then it has proof that the majority of the replicas agree on the ordering in which the request should be processed. The replicas however, do not know that such agreement quorum exists – they only know about their local execution.
   A problem that may arise at this point is when a *view change* occurs. Since the view $v + 1$ must be consistent with the state in view v the *view change* subprotocol must know which requests were executed in view v. Since up to f replicas may be faulty the *view change* subprotocol must use information from a quorum of $2f + 1$ replicas.
   To solve this problem the client sends a message containing a *commit certificate* that contains a list of $2f + 1$ replica signed-portions of the response given to the client and the corresponding $2f + 1$ replica signatures.
   Replicas then receive the message from the client containing the *commit certificate* and acknowledges its reception to the client. The replica upon receiving the *commit certificate* can validate if the request should be executed in the current view (since the request has its own sequence number and history). Note that the request was already executed by a correct replica so it shouldn't be executed again to respect exactly-once semantics.

2. The client then only needs to receive the confirmation by $2f + 1$ replicas to consider the execution to be completed. Since the *view change* subprotocol needs $2f + 1$ replicas for execution we know that at least $f + 1$ replicas are definitely correct and stored the commit certificate.

The *view change* case occurs when the client receives fewer than $2f + 1$ responses or the client suspects a faulty primary.

When the client receives fewer than $2f + 1$ matching responses it resends the request message to the replicas (so they can track the progress and possibly initiate a view change). Upon receiving the request from the client, a replica checks if the request has a higher timestamp than the chached response for that client and then sends a message to the primary server while initiating a timer. If the replica receives a response from the primary then it proceeds with the protocol as described earlier. If the time runs out then the replica starts a view change. To maintain exactly-one semantics, all replicas have a cache of the last reply for each client.

If the client receives valid response messages from the same request but with different sequence numbers or history, then it is declared as a Proof of Misbehavior [15] on the primary. Noticing this, the client sends a message to all replicas with the Proof of Misbehavior and a view change will initiate.

3.4.4 Aardvark Since PBFT, many systems have appeared that promised high throughputs under Byzantine conditions (Q/U, HQ and Zyzzyva). However the high throughputs were often reached by relaxing some conditions – Zyzzyva moved many responsibilities to the client for the recovery of a system whenever a fault was detected. Relaxing such conditions can render a system unavailable for long periods of time – the system is *fragile* to faults.

The basic idea of Aardvark is to build a protocol that offers high throughput while maintaining robustness so that the system offers predictable performance (even under faults). Aardvark however deviates from some of already established "conventional wisdom" – it uses signatures instead of MACs, despite being considered an important performance bottleneck [3], performs regular view changes even when they may disrupt the service temporarily and the use of point to point communication instead of IP-multicast.

The authors define two important concepts that describe the execution conditions:

- Gracious Execution: an execution is considered gracious if it is synchronous (known short timeout on a message delay) and all clients and servers are correct.
- Uncivil Execution: an execution is considered uncivil if it is synchronous (known short timeout on a message delay) and up to f clients and servers are byzantine.

Aardvark rejects any optimizations while the system is under a *Gracious Execution* that may degrade the performance of the system while in *Uncivil Ex-*

ecution. The system also operates under an asynchronous network where *Synchronous Intervals* occur. *Synchronous Interval* is a time interval where any message sent by any correct process (client or server) is delivered within a defined time bound.

The main contributions of Aardvark are also the three main core differences between previous BFT systems: signed client requests, resource isolation and regular view changes.

- The usage of signed client requests comes from the property of non-repudiation and ensures that all correct replicas validate the client request identically thus "eliminating a number of expansive and tricky corner cases found in existing protocols that make use of a weaker (though faster) message authentication code (MAC) authenticators" [7] – "one node validating a MAC authenticator does not guarantee that any other nodes will validate that same authenticator". While signed requests are expensive, they are only used to sign client requests and this is done on the client side leaving the servers with this expense. Since cryptographic signing requests is a an expensive operation an attacker could send a large number of requests that need to be verified on the server side. To limit this, Aardvark puts a hard limit on the number of incorrect/faulty signatures that a client can send to the system (by using a hybrid MAC-signature) and it also forces the client to complete one request before sending another one.
- For resource isolation, Aardvark deploys unique Network Interface Controllers (NICs) – each replica has a one-to-one channel that is used for message exchange. This allows each replica to defend itself against attacks (by disabling the attack source NIC). It also prevents a faulty replica to interfere on the message delivery by correct replicas. However, by having unique interfaces for communicating between pairs of replicas, the system suffers from a performance hit – it does not use multicast to optimize all-to-all communication. To optimize the usage of each interface, Aardvark differentiates client requests with the communication between individual replicas by using different work queues. By doing this, Aardvark prevents client requests to interfere on replica-to-replica communications.
- Aardvark also changes views regularly. Replicas monitor the performance of the current primary server and continuously raise the acceptable throughput level. When the primary fails to correspond to the current level, replicas start a view change. View changing was treated as an expensive protocol and only used when the throughput was dropping fast or other extreme situations. However the view change protocol "is similar to the regular cost of agreement". While doing a view change, the system cannot process new requests but performing a view change only when we have a faulty primary is more costly than regularly changing the view [7].

The steps performed by the protocol follow a standard three phase commit protocol [3] that was already explained in Section 3.4.2. However, the steps are performed using the three main core differences already explained.

As described, replicas frequently monitor the performance of the primary server: by slowing increasing the throughput rate which the primary must satisfy. Moreover, Aardvark expects from the primary a frequent supply of PRE-PREPARE messages and high, sustainable throughput. A timer is set and whenever a PRE-PREARE message is issued the timer resets. If the timer expires, a view change is initiated. There are also periodic checkpoint where replicas assess the throughput of the system. If the performance drops below a certain threshold then a view change is initiated.

Another issue is that a faulty primary, to avoid being replaced, may issue many PRE-PREPARE messages. However it will be caught on the periodic checkpoint where replicas will check the current throughput of the system (completed requests) and, therefore, the primary will face a demotion.

4 BFT Reconfiguration

As we have described in the previous section, BFT protocols are relatively expensive and require the exchange a significant amount of messages during their execution. Also, their performance depends on operational context: some protocols are more efficient in stable runs and others offer better performance under attack. Furthermore, the number of messages exchanged depends on the number of processes that exist in the system which is often a function of the number of faults that need to be tolerated. Finally, as in any fault-tolerant protocol, nodes may fail and need to be replaced. Since many of the factors that affect the performance of a BFT protocol may be hard to estimate *a priori*, and may change dynamically, it is relevant to consider the design and implementation of BFT protocols that can adapt in run-time.

When considering the design of adaptive BFT protocols, there are three different aspects that need to be considered:

- *What to adapt,* i.e., what are the possible system configurations that may selected. In this work we mainly consider four different adaptations: integration of new replicas (or re-integration of replicas that have recovered) to replace failed replicas, dynamic changes to the number of replicas, migration of replicas to containers to avoid recently uncovered vulnerabilities, and dynamic changes to the BFT protocol itself.
- *When and why to adapt,* i.e., what are the policies that drive the selection of the target configuration of the system given a system state. In this aspect there are two facets to consider. One is where the adaptation code is executed: it can be embedded in the BFT protocol (what we call a *monolithic approach*) or can be implemented in a separate module that is in charge for the reconfiguration (what we call a *modular approach*). To achieve modularity, a BFT implementation may provide an interface exposing methods to allow its reconfiguration, thus separating the reconfiguration mechanisms from the rules that trigger that same reconfiguration [16]. The other aspect is to distinguish implementations where the system is only able to execute a

fixed set of policies, that are hard coded in the implementation from implementations that are flexible enough to implement a variety of policies. Typically, modular implementations are more amenable to implement a richer set of policies.

– *How to adapt,* i.e., what algorithms are used to perform the system reconfiguration. While, in principle, it is possible to reconfigure a system by first halting its operation, and restarting it under a new configuration, such approach is not very appealing given that it may create long periods of service unavailability. Therefore, reconfiguration algorithms attempt to support the dynamic reconfiguration with minimal impact on the performance of the system.

In the following paragraphs, we address each of these aspects with more detail.

4.1 What to Adapt

In this work, we focus on the following set of adaptations that can be applied to a BFT implementation:

1. *Replica reintegration*: As any other fault-tolerant protocol, a BFT deployment can only tolerate a fixed number of faults at a given time. To ensure that the system will keep operating after failures occur, it is therefore fundamental to replace replicas that have failed, by correct replicas. The new replicas may be completely fresh or may already contain some some state that has been collected by that replica or by some other replica before the crash. In any case, the new replica will never be completely up-to-date (with regard to the state of active replicas) and some re-integration protocol must be executed to put its state on par with the current active correct replicas. Also, we have seen that many BFT implementations keep a *view* of what replicas are correct; therefore, replica reintegration typically involves the execution of the view change sub-protocol.
2. *Changing the number and/or location of replicas*: The replica re-integration process described above can be seen as a particular case or a more general adaptation that consists in the process of adding, removing or migrating replicas in a BFT deployment. Changing the number of replicas in runtime allows to scale the resilience of the system according to the estimated level of threat. In particular, the number of replicas may be increased if the risks of having machines compromised is higher and lowered when the risks decrease. Adjusting the number of replicas is important because, as we have seen, in BFT protocols there is an inherent trade-off between the resilience and the performance of the deployment. Replica migration can be helpful if a machine (where a replica is running) lacks resources or has some vulnerability that has been exposed.
3. *Changing the BFT protocol*: As we have seen, different BFT protocols exist and none of them offers the best performance in all scenarios. On the

contrary, some perform better in stable scenarios and others before better when the system is unstable. Since in a real deployment the environmental conditions may change in runtime, it is interesting to support the dynamic adaptation of the BFT protocol in use, in order to execute the protocol that is more suitable for the observed conditions. The biggest challenge in changing the BFT protocol is to coordinate the replicas such that the properties of the service are guaranteed during the change. The most straightforward manner of achieving this is to stop one protocol at all replicas before activating the execution of the new protocol. This may not be trivial, because one needs to assure that different protocols do not make inconsistent decisions about the same message. For that, existing BFT protocols need to be adapted/extended in order to support the concept of *abortability* and correctly maintain correctness when transitioning from *State A* to *State B*.

4.2 When and Why to Adapt

In this work we are interested in building modular solutions to drive the adaptation. Therefore, instead of supporting a fixed set of policies and protocols that are hardcoded in a monolithic BFT implementation, we aim at a solution where there is a separate adaptation manager that can use a number of actuators to drive the reconfiguration of the BFT implementation. The adaptation manager can be seen as logically centralized component that monitors the system operation, extracting metrics regarding the observed performance (bandwidth, memory usage, number of clients, etc...) but also information about system vulnerabilities and level of threat (for instance, resorting to intrusion detection systems). Using this information, the adaptation manager uses adaptation policies to select if and when the system must be reconfigured. The choice of the correct adaptation can be driven by user-specified rules (for instance event-condition-action rules[10]) or can be performed using some automated manner (for instance, resorting to machine-learning[17].

We now make a brief overview of some of the policies that have been proposed in the literature, for the adaptations listed in the previous section:

- Castro and Liskov [18] suggest that replica re-integration should be performed not only to replace replicas that have failed but also as a *proactive* measure of defense against replicas that may have been compromised, even when they remain asymptomatic. The idea is that, in most cases, it takes a reasonable amount of time for an adversary to compromise a machine. Therefore, if the machine is reboot and a fresh copy is re-installed, this may foil the efforts of an attacker and effectively prevent the adversary from taking more that 1/3 of the replicas. This policy became known as *proactive recovery*.
- The BFT-SMaRt system (described in Section 5) has a module for reconfiguration which allows replicas to be added or removed from the system. The paper also describes the tradeoffs involved in this reconfiguration. With more replicas, the system can tolerate more arbitrary faults (at the cost

of more messages going through the network). Since adding replicas has a cost associated with it (state transfer and view change), we should only add replicas when strictly needed.

– Abstract [10] is a system that provides interchangeability between different BFT protocols. The paper focuses on the mechanisms required to replace the BFT implementation but also suggests some simple policies to trigger a switch. These policies are static and hard-coded in the BFT implementations. The paper proposes to switch from a protocol that has good performance with favorable scenarios to a more expensive but also more robust protocol when the primary is unstable. Switching happens when the primary is suspected of being malicious/faulty and, consequentially, the system remains in the more expensive configuration for a quarantine period before it reverts back to the optimistic protocol.
– Adapt [17] further explores this idea by introducing an evaluation step for electing the next active BFT protocol. Adapt is an adaptive abortable BFT system that reacts to changing conditions of the environment while introducing Machine Learning techniques for the protocol selection process. Adapt uses an adaptation manager that has three operating modes: *static*, *dynamic*, and *heuristic*. The *static* mode only analyses the default characteristics of each protocol (we can say that they are the static features that represent the protocol such as the number of replicas needed to tolerate f faults). The evaluation process in *static* mode is performed before the system starts: the protocol chosen in this this step will be the starting protocol. In *dynamic* and *heuristic* modes, the evaluation process is done in run-time while evaluating the performance of the active protocol under the conditions in which the system is present. Moreover, the *heuristic* mode uses heuristic rules for the evaluation step. Adapt also uses two performance metrics for the evaluation process. The *Key Characteristic Indicators* (KCI) represent the static features of a given protocol (toleration of client faults, number of replicas, etc...). The *Key Performance Indicators* (KPI) are metrics that are dynamically computed (using prediction methods) and represent the performance of the protocol in run-time (such as progress and throughput). The KPI metrics are computed experimentally and obtained using techniques like Support Vector Machines or Regression [17] – each protocol runs a period of time, while the *impact factors* change in order to get the KPI values under each state. After collecting a set of KPI values, we have a training set that is used to train a *prediction function*. The *prediction function* takes as input the *impact factors* and outputs the corresponding KPI value. In order to improve the *prediction function*, the training set is also updated while the Event System (ES) sends new events allowing the function to tune itself.

One important factor that needs to be taken into account is the cost of protocol switching since it has an associated overhead. After the selection of a new configuration by the adaptation manager, we know that it will be the best protocol present in the library given the new conditions. But if the benefits of switching are not significant to the performance, then the system would suffer

the cost of protocol transition without gaining too much from that same cost. Therefore, the Adapt[17] system uses a *switching threshold*, defined as follows:

$$\frac{p_{max}}{p_{curr}} \geq S_{thr}$$

Where p_{max} is the score for the best protocol chosen by the evaluation process, p_{curr} is the score for the currently active protocol and S_{thr} is the threshold value that can be defined by administrators.

Note that the adaptation manager itself can be a target of malicious attacks so some solutions require that the reconfiguration module is also replicated among the existent replicas. Thus, the replication adaptation manager will also need to execute an agreement protocol to decide on the reconfiguration result. Given that the adaptation manager only needs to agree on reconfigurations sporadically, the performance of the BFT protocol used to replicate the adaptation manager is not as critical as the performance of the BFT protocol supporting the application. Therefore, any robust BFT protocol may be used to coordinate the replicas of the adaptation manager.

4.3 How to Adapt

We now discuss some of the challenges involved in performing the adaptations listed in Section 4.1.

4.3.1 Replica integration. One of the most heavy-weight tasks associated with adding or re-integrating a replica is the process of bringing the replica up-to-date, typically by transferring state from other active replicas. Several strategies have been proposed to increase the efficient of this process, including:

- *Incremental Transfer*: hierarchical state transfer and incremental cryptography [18].
- *Parallel Logging*: this is achieved by logging groups of operations instead of single operations and by executing the operations in parallel with their storage [19].
- *Sequential Checkpointing*: the premise for this idea is that, by doing a synchronized checkpoint (all replicas checkpoint at the same time) the system cannot make any progress. Thus since just a quorum of replicas is needed to make progress, replicas perform a checkpoint in a sequential fashion [19].
- *Collaborative State Transfer*: the state transfer is performed in a collaborative way with each replica contributing to the replica recovery [19].

4.3.2 Changing the underlying protocol. Abstract [10] is an adaptive BFT system that proposes a protocol to implement a safe transition between two distinct BFT implementations. Abstract is based on the concept of *instances*. Instances are like a replicated state machine service and operate like so (they have a set of replicas that receive the clients' requests, execute them and return the

result to the client). Each reply to the client contains a commit history which has a sequence totally ordered of client requests. An important concept introduced in Abstract is the concept of *abortability* – if some *progress* conditions are not met while executing a request, the request is aborted. These *progress* conditions are determined by the system administrator and the task of setting what is considered progress is facilitated: the administrator only needs to establish in which conditions the system makes progress instead of designing a system that tries to make progress under all conditions. Abstract allows instance reconfiguration like traditional state machine reconfiguration [16] but the difference is that the reconfiguration is performed on top of *abortable* state machines. Each instance has a unique identifier (instance number), a number of replicas that implement the service and the protocol being used (along with other internals such as the current view number, primary, etc...).

The reconfiguration protocol of Abstract can be described in three different steps:

1. Stopping the current Abstract instance: an abstract instance is immediately stopped as soon as it aborts a single request. Along with the abort signal, the instance also returns an *abort history* that contains a mapping between the replica's identification number i and its commit history.
2. Choosing the next Abstract instance: along with the abort signal, the aborting Abstract instance i also returns the identifier of the next instance $next(i)$. As with consensus, the result for $next(i)$ should be the same across all abort indications of instance i. Other property is that $next(i) > i$.
3. Combining the commit histories: the client uses the abort history of instance i in the invocation of $next(i)$. This will be referenced as the *init* history of the instance $next(i)$. The *init* histories are used to initialize an instance before it starts its own execution (accepting and executing clients' requests).

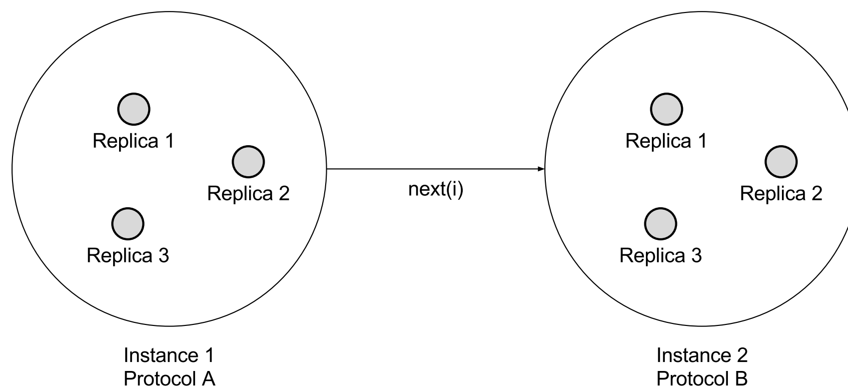

Fig. 7. Abstract with two configured instances

Abstract also provides three important properties:

1. Instance switching is idempotent: the composition of two Abstract instances yields another Abstract instance.
2. A correct implementation of a Abstract instance always preserves the safety conditions of a state machine (such as the total order of committed requests)
3. A replicated state machine is nothing but a special Abstract instance – one that never aborts

One important rule is required while running Abstract – there is only one active instance in the system and that instance is the only one that can commit requests. As with many other system models, Abstract assumes that that links between are asynchronous and unreliable and up to f processes can fail arbitrarily (byzantine processes) and the total number of replicas present in the system n is $3f + 1$. A strong adversary may coordinate faulty nodes but cannot break the cryptographic assumptions of hashing, authentication codes and signatures. Abstract also assumes that synchronous periods occur allowing the processes to communicate in a timely fashion (there is a known upper bound t for message transmission by correct processes). With this model, an administrator can pick different Abstract instances (each one representing a different protocol) and decide in which conditions each instance runs.

In the Abstract paper, these mechanisms are illustrated by performing dynamic adaptation between two BFT protocols, one derived from Zyzzyva[4], namely ZLight that captures Zyzzyva's fast case and another called Backup that is based on PBFT. The Backup instance only works on top of a valid init history (one produced by a valid ZLight instance) – upon receiving a valid commit history, the instance state is reconstructed by executing every request provided by the history. After the state restoration, Backup commits exactly k requests and, after the execution of the kth request, it aborts and sends its commit history (digitally signed) back to the client. The choice for the value of k is up to the system administrators but it is important to know that it must be a value that contributes to the progress of the system – it should allow the Backup instance to be active for a long period of time (since we are in a state to handle failures) but also short enough so that it does not confine clients to the Backup instance for too long (since ZLight is capable of doing more progress under the "common case" conditions).

Adapt [17] further explores the ideas above by introducing an evaluation step for electing the next active BFT protocol. Adapt is an adaptive abortable BFT system that reacts to changing conditions of the environment while introducing Machine Learning techniques for the protocol election process. Adapt offers three main benefits over an Abstract implementation:

1. No protocol order has to be defined a priori: the way that Abstract changes between instances needs to be already known before deploying the system. In AZyzzyva, the system was designed to change between the ZLight instance and the Backup instance – the two instances were known beforehand as the conditions that would trigger the switch. In Adapt we are not constrained by the number of protocols or by which order the instances change.

2. Switching can occur when we can gain performance or increase progress: Adapt is not constrained by a set of conditions in which the protocol change only occurs when there is a failure – if a performance boost can be achieved by switching the active protocol, then that change will be applied.
3. Fallback mechanisms are no longer needed – AZyzzyva has Backup instance as a fallback mechanism to guarantee that progress is being made while the "common case" conditions are not met. In Adapt the change occurs when there is a more appropriate protocol for the new conditions and no fallback mechanism is needed if those conditions maintain.

5 BFT Smart

In this Section we will discuss BFT-SMaRt [8], a modular system that implements a BFT protocol similar with PBFT [3]. BFT-SMaRt will be the starting point for this project and its our goal to provide an extension to its underlying architecture allowing the possibility of reconfiguring the system in run-time automatically (i.e. without the user's intervention) (see Section 2). The BFT-SMaRt will be extended as is described in Section 6.

5.1 The Modular Approach

BFT-SMaRt [8], is a system that implements Byzantine Fault Tolerance in SMR on top of Java Virtual Machine (JVM). The main contribution of BFT-SMaRt is its modular approach to a practical implementation of a BFT system while documenting and providing modules that enable state transfer and replica reconfigurability. By suggesting a modular approach, BFT-SMaRt steps away from the *monolithic* approaches (like PBFT [3]) and, additionally, provides a simple and extensible programming interface.

BFT-Smart implements different modules each one for a specific task within the system such as consensus, reliable point-to-point communication, client requests ordering, state transfer and reconfiguration. It does so while also providing specific interfaces that enable the communication between those modules.

It follows the classic SMR approach where clients send requests (through the *invoke(command)* method) and replicas execute the requests (through the *execute(command)* method). Due to its modular and extensible nature, it is possible to implement different behaviors with different calls/callbacks or via plug-ins both at client and server side.

The core protocols used are by BFT-SMaRt are:

- Total order multicast: this protocol is achieved by the module Mod-SMaRt. Mod-SMaRt uses consensus to achieve total order. Mod-SMaRt has two phases: normal and synchronization. In the normal phase there are no faults and the system is in a period of synchrony. When these conditions are not achieved, Mod-SMaRt switches to the synchronization phase where the election of a new leader takes place and replicas make the transition to the new

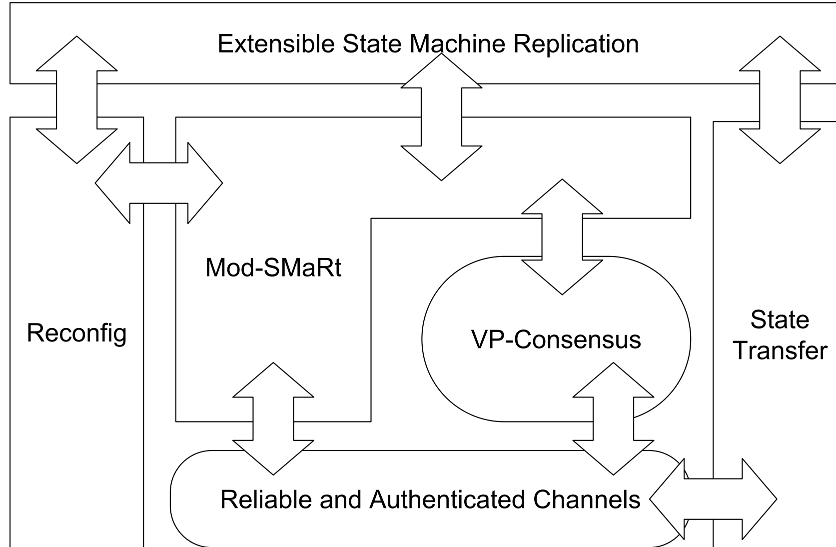

Fig. 8. The architecture of BFT-SMaRt

consensus instance (which may use the state transfer protocol). Mod-SMaRt is described in depth in [20].

- State Transfer Protocol: this protocol allows replicas to be reintegrated in the system without restarting it. It is based on the ideas presented in [19] mentioned in Section 4.3.1.
- Reconfiguration Protocol: this protocol allows replicas to be added or removed on run-time and is only accessible by system administrators. These actions can only be initiated by instances of the *View Manager* which then notifies the Mod-SMaRt of the replica to be added/removed. The operation of adding and removing replicas is totally ordered so all correct replicas will have the same view (the view of the system). These special operations of adding and removing replicas must have been signed cryptographically by the system administrator. When a replica receives these special operations, it validates the signature – if it is valid, the current view is updated and a confirmation is sent to the View Manager. Once the View Manager receives the operation result confirmation it notifies the added/removed replica to start/stop its execution. This protocol also triggers the State Transfer Protocol because a newly added replica must be on par with the history of all the other correct replicas. Finally, all clients must store the current view of the system – any request that is performed in an old view is rejected but, upon rejection, the client receives the latest view so it can retransmit its operation while acknowledging the system's current view.

BFT-SMaRt is our starting point for the development of a dynamic byzantine fault tolerant system – its API and modular approach eases the extension and creation of new modules for additional functionality. The following section describes how the system will be extended – which modules will be added and why.

6 Architecture

The architecture of the system will be based on the architecture present in BFT-SMaRt. The BFT-SMaRt project (available in [21]) offers a modular SMR approach that eases the implementation of new features (which can be problematic in monolithic approaches). Moreover, to support the dynamic selection of the running protocol, the Mod-SMaRt will need to be extended or modified. Remember that Mod-SMaRt is the module responsible for the implementation of the properties of state machine replication and for the communication pattern between replicas.

The concept of *abortability* (that was also introduced in Abstract [10] and further explored in Adapt [17]) will be introduced in BFT-SMaRt to ease the transition of one state to another.

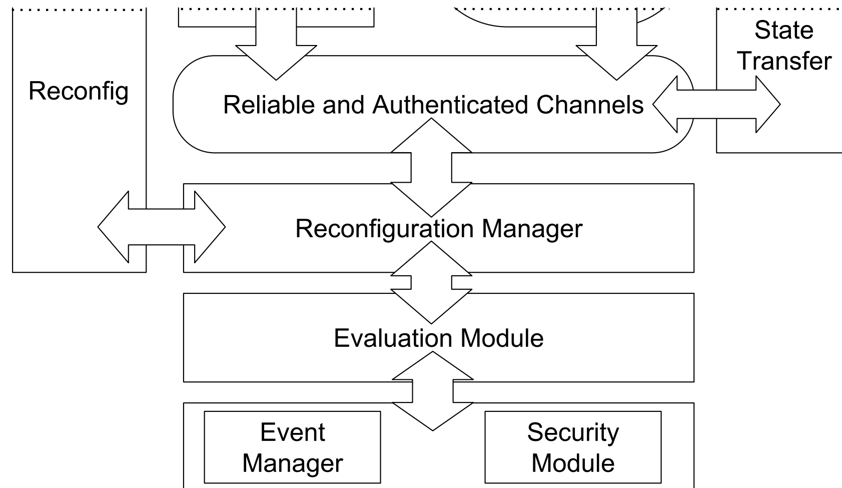

Fig. 9. Reconfiguration Manager, Event Manager, Evaluation and Security Modules will be the new sub-systems added to BFT-SMaRt

The following modules will be added to BFT-SMaRt:

- Event Manager and Security Module: these two components will handle the events that occur within the environment. Those events represent changes to

the system such as bandwidth, memory usage, system progress, among others. The event manager will watch the changes for a set of metrics (described in Section 7). The Event Manager then notifies the Evaluation System. A Security Module will also be added in order to detect potential malicious patterns on the watched metrics.
– Evaluation Module: this module will trigger an evaluation process to assess the possibility of a needed reconfiguration (protocol change, add/remove replicas). If a change is required then the system is notified about that given change. The evaluation module will also take into consideration the cost of performing the reconfiguration needed. The situation where an attacker constantly triggers the conditions necessary to trigger a reconfiguration will be also considered in this module – if the system is always performing reconfigurations, it greatly impacts its progress.
– Reconfiguration Manager: this module is responsible for performing the reconfigurations on the replica. It receives the commands from the Evaluation Module and executes them – there is no evaluation done in this module (all the evaluation steps are executed in the Evaluation Module and a command is given as a final decision). It communicates with the Reconfiguration System already present in BFT-SMaRt for adding or removing replicas. For the protocol change, it aborts the current active protocol and notifies the current active replicas to switch to the new one.

In order to understand the roles of the new modules we present three use cases that trigger a reconfiguration:

– *Change the underlying BFT protocol*: the objective of changing the active BFT protocol is to give the best possible *progress* (execution of client requests) while maintaining the correctness of the system. If the system is running a protocol that offers high throughput of client requests but under specific optimal conditions (like Zyzzyva's Fast Case) and a malicious attack is performed on the system where the bandwidth is highly affected, then we need to guarantee that *progress* continues to be made. By switching to a more robust solution (like PBFT or Aardvark) we allow the system to make progress even when we do not have the optimal conditions needed for Zyzzyva's Fast Case. A question remains however: Why perform a switch when a system like Zyzzyva offers a fallback mechanism when the Fast Case conditions are not met? Remember that Zyzzyva always tries to perform the Fast Case scenario (and additional steps are always performed if the system is under faults). A malicious attack can affect a system for long periods of time and a more robust solution is more adequate to those scenarios – one that does not always tries to perform its best case scenario even when the necessary conditions are not met. So when a malicious attack occurs, the Security Module will warn the Evaluation Module of the metrics affected and their current condition. The evaluation module then analyses the reconfiguration that offers the best performance under the new conditions and sends a command, for example, to change the running BFT protocol. The

Reconfiguration Manager receives the command, aborts the current protocol and loads the one issued by the Evaluation Module.

- *Change the number of replicas*: Changing the number of replicas is already supported by BFT-SMaRt in the Reconfiguration module. BFT protocols usually assume a a number of replicas $n \geq 3f + 1$. Although the Security Module captures malicious patterns, in a possible attack scenario, a system administrator can increase the number of replicas at will in order to support more faults even when those patterns are not detected. However, by increasing the number of replicas, the number of messages in the network also increases so, when the system is considered stable under the current conditions, replicas can be removed in order to decrease the bandwidth usage, while maintaining stability. The manual reconfiguration is already offered by an interface of BFT-SMaRt. Additionally, with the integration of a security module, we hope to dynamically reconfigure the number of replicas without the need of any manual intervention.
- *Change the replica's Operating System*: BFT-SMaRt runs on top of the Java Virtual Machine which means that is possible to run replicas under a different Operating Systems (which support the JVM). This allows to restart a replica to a different OS if a vulnerability is found in the OS. Along with the metrics registered, we can check which OS the replica is running and, add or remove replicas with the OS that improves the performance of the system while maintaining a "pool" of different Operating Systems running in order to provide heterogeneity (the greater the variety of running operating systems, the more robust the system becomes against malicious attacks).

7 Evaluation

In order to evaluate if the project goals have been achieved, an additional BFT protocol will be implemented under the BFT-SMaRt architecture. The following metrics will be analyzed: bandwidth, memory usage, number of clients, number of replicas and throughput (number of client requests processed).

These metrics will be analyzed under different three different system reconfigurations (that are described in Section 6), more specifically under the following combinations:

1. Maintain the number of replicas and operating system but change the active protocol.
2. Maintain the number of replicas and active protocol but change the operating system.
3. Maintain the active protocol and operating system but change the number of replicas.

Moreover, these configurations will be evaluated under a stable execution and when a malicious attack is performed on the system. The system should always maximize the progress being made despite the present conditions, while maintaining correctness.

8 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.

9 Conclusions

Throughout the years we have seen many protocols that tackle the Byzantine faults problem. PBFT [3] initiated an "arms race" of byzantine protocols that were looking to offer high throughput in a practical system. Solutions have appeared, from more fragile solutions that offer high throughput [4] to more robust ones that guarantee progress at the cost of throughput and communication steps [7].

In more recent years, and with a collection of BFT protocols in hand, works like [10] and [17] offered a dynamic solution – the system designers were no longer restricted to one specific protocol that offered a given throughput under specific conditions – instead, these ideas were adapted to work with the concept of *abortability* where a given active instance may abort the current execution and pass control to another new instance that is more appropriate to deal with the new scenario.

Furthermore, as a mean to step away from monolithic design, BFT-SMaRt provides a modular approach that offered an extensive interface that allows intermodule communication. The system also runs on the Java Virtual Machine giving the possibility for it to run on a variety of different operating systems. Exploring this modular approach, we will adopt some of the ideas introduced in Abstract [10] and Adapt [17] in order to provide the system the ability to adapt to new conditions.

Acknowledgments We are grateful to R. Rodrigues and M. Correia for the fruitful discussions and comments during the preparation of this report.

References

1. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565
2. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3) (1982) 382–401

3. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: OSDI. Volume 99. (1999) 173–186
4. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: ACM SIGOPS Operating Systems Review. Volume 41., ACM (2007) 45–58
5. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. ACM SIGOPS Operating Systems Review **39**(5) (2005) 59–74
6. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association (2006) 177–190
7. Clement, A., Wong, E.L., Alvisi, L., Dahlin, M., Marchetti, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: NSDI. Volume 9. (2009) 153–168
8. Bessani, A., Sousa, J., Alchieri, E.E.: State machine replication for the masses with bft-smart. In: Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, IEEE (2014) 355–362
9. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to reliable and secure distributed programming. Springer Science & Business Media (2011)
10. Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 bft protocols. In: Proceedings of the 5th European conference on Computer systems, ACM (2010) 363–376
11. Lamport, L.: The implementation of reliable distributed multiprocess systems. Computer Networks (1976) **2**(2) (1978) 95–114
12. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2) (1988) 288–323
13. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2) (1980) 228–234
14. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2) (1985) 374–382
15. Aiyer, A.S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.P., Porth, C.: Bar fault tolerance for cooperative services. In: ACM SIGOPS Operating Systems Review. Volume 39., ACM (2005) 45–58
16. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. ACM SIGACT News **41**(1) (2010) 63–73
17. Bahsoun, J.P., Guerraoui, R., Shoker, A.: Making bft protocols really adaptive. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE (2015) 904–913
18. Castro, M., Liskov, B.: Proactive recovery in a byzantine-fault-tolerant system. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, USENIX Association (2000) 19–19
19. Bessani, A.N., Santos, M., Felix, J., Neves, N.F., Correia, M.: On the efficiency of durable state machine replication. In: USENIX Annual Technical Conference. (2013) 169–180
20. Sousa, J., Bessani, A.: From byzantine consensus to bft state machine replication: A latency-optimal transformation. In: Dependable Computing Conference (EDCC), 2012 Ninth European, IEEE (2012) 37–48
21. : Bft-smart code project. https://github.com/bft-smart/library Accessed: 2015-12-06.