Monitoring Byzantine Fault Tolerant Systems to Support Dynamic Adaptation

Bernardo Palma
bernardo.palma@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. An adaptive system is a system capable of altering its configuration in response to changes in its execution environment. A key component of any adaptive system is a monitoring infrastructure, that is able to collect information regarding the system operation and feed the policies that guide the system adaptation. In this work, we describe the first steps to build a monitoring infrastructure for supporting the dynamic adaptation of Byzantine fault-tolerant systems. Building such monitoring system encompasses two challenges: identifying i) what is the relevant information that should be monitored, and that may be useful to manage the adaptation, and; ii) the mechanisms that allow to provide this information, in an efficient, consistent and fault-tolerant manner to an adaptation manager that also needs to be replicated. This report covers both of these aspects.

Table of Contents

1 Introduction . . . 3
2 Goals . . . 4
3 Byzantine Fault Tolerance . . . 4
3.1 BFT Protocols . . . 4
3.1.1 PBFT . . . 4
3.1.2 Zyzzyva . . . 6
3.1.3 Quorum and Chain . . . 8
3.1.4 Discussion . . . 9
4 Adaptive Systems . . . 9
4.1 State Machine Replication . . . 10
4.1.1 SMR's Parameter Reconfiguration . . . 12
4.1.2 Fault Model Reconfiguration . . . 12
4.1.3 Protocol Reconfiguration . . . 13
4.1.4 Replica Adaptations . . . 13
5 State Machine Replication System Monitoring . . . 13
5.1 Monitoring System's Architecture . . . 14
5.1.1 Fault Tolerant Monitoring System . . . 14
5.2 Relevant Metrics to Monitor . . . 15
5.3 Fault Detection . . . 16
5.3.1 Background . . . 17
5.3.2 Interest in Fault Detection . . . 18
6 Architecture . . . 20
6.1 Overview . . . 20
6.2 Sensors . . . 21
6.3 Aggregator . . . 21
6.3.1 Aggregation Function . . . 23
6.3.2 Prediction Function . . . 23
6.3.3 Metric Propagation . . . 24
6.3.4 Extensibility . . . 24
7 Evaluation . . . 25
8 Scheduling of Future Work . . . 26
9 Conclusions . . . 26

1 Introduction

A Byzantine Fault Tolerant system (BFT) is a system able to tolerate arbitrary faults[1]. Byzantine fault tolerance is inherently expensive: in the general case $3f + 1$ replicas are needed to tolerate f Byzantine faults. Unfortunately, the likelihood of systems suffering arbitrary faults tends to increase due to a combination of technological factors (such as the increase of density in chips) and social factors (such as the increased risk of malicious attacks). Therefore, the design and development of BFT systems has gained significant interest in the last decades[2–8].

As a result of the intense research in the field, several different BFT protocols have been developed, each optimized for different operational conditions. This has opened the door for the construction of adaptive BFT systems that can dynamically commute among different protocols, in response to changes in the environment[7, 9].

A key component in any adaptive system is a monitoring infrastructure, that is able to collect information regarding the operation envelope of the managed system. Information regarding the available resources (CPU, network bandwidth), the system workload (rate of requests, profile of requests), occurring faults, vulnerability to attacks, can be used by the polices that drive the system adaptation, to select the most appropriate protocol configuration to achieve some predefined system goals.

In a Byzantine fault-tolerant system, the monitoring system must be, itself, tolerant to Byzantine faults. This means that the monitoring system must be built using redundant components, such that a single fault will not cause inaccurate or inconsistent information to be passed to the components that control the system adaptations.

In this report, we address the design and implementation of a monitoring infrastructure for supporting the dynamic adaptation of Byzantine fault-tolerant systems. In this context, we are interested in addressing two complementary aspects, namely: identifying i) what is the relevant information that should be monitored, and that may be useful to manage the adaptation, and; ii) the mechanisms that allow to provide this information, in an efficient, consistent, and fault-tolerant manner to an adaptation manager that also needs to be replicated.

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 the background research related with our work. More specifically, in Section 3 we discuss *Byzantine Fault Tolerance*, showing some of the different existent state of the art protocols, in Section 4 we present what is an *Adaptive System*, how its composed and go into detail on our managed system, finally in Section 5 we will discuss *System Monitoring*, why it is needed and what is important to collect from the environment. 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 designing and implementing a monitoring infrastructure for supporting the dynamic adaptation of Byzantine fault-tolerant systems. More precisely:

> *Goals:* We aim at designing a redundant monitoring infrastructure that can provide consistent and accurate information regarding the operational environment to the replicas of a fault-tolerant adaptation manager, in order to feed the polices that drive the adaptation of a Byzantine fault-tolerant system.

To achieve this goal we will need to design a monitoring architecture that can support the availability of replicated sensors when collecting system information. Data collected from these multiple sensors needs to be aggregated and processed to filter inaccurate or inconsistent data that may be provided by faulty sensors. Ideally we would like to derive an extensible monitoring infrastructure, that could be easily augmented with new sensors and new fault-tolerant aggregation functions in a seamless manner. A prototype of the monitoring system will be implemented and integrated in the BFT-SMaRt[10] framework.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the monitoring infrastructure; ii) an implementation of the monitoring system, iii) an extensive experimental evaluation using the resulting prototype.

3 Byzantine Fault Tolerance

3.1 BFT Protocols

In this Section we will present some of the state of the art protocols for byzantine fault tolerance, namely for state machine replication, and show the characteristics and differences between them. Although state replication protocols usually have three sub-algorithms, agreement, view-change, and checkpointing, we will focus on the agreement protocol as it captures the main differences among existing systems.

3.1.1 PBFT The protocol known as Practical Byzantine Fault Tolerance (PBFT)[2] is one of the first BFT protocols designed with the aim of offering acceptable performance while being able to operate in asynchronous environments. Its design has been highly influential and spurred several research projects to derive other efficient implementations of BFT protocols. PBFT supports state machine replication[11]. It requires at least $3f + 1$ replicas to tolerate f faulty replicas. During PBFT operation, replicas have different roles: one replica is designated the primary and the others are designated the backups. A given configuration, with roles assigned to each replica is called a view. If the primary is

suspected to be faulty, the system moves to another configuration, in a process that is called a view change. The evolution of the system is characterized by a sequence of views. In the most frequent case, the system operates as follows: the client sends a request to the primary replica, that triggers a three-phase protocol to atomically multicast the request to the rest of the replicas. This protocol is composed of the following phases:

1. *Pre-prepare phase:* The primary assigns a sequence number to the request received (if the primary is still processing a previous request, it will buffer incoming requests, later processing all of them in batch, using a single agreement) and multicasts a PRE-PREPARE message to all the backup replicas (the request is piggybacked in that message).
2. *Prepare phase:* A backup replica checks the PRE-PREPARE message and, if it is considered valid, the replica multicasts a PREPARE message to all other replicas. The participation of the backups is required to ensure the total order of requests sent in a given view, even when the primary is faulty. Each replica then needs to receive a valid PRE-PREPARE message and at least $2f$ matching PREPARE messages from different backups, in order to move on to the next phase.
3. *Commit phase:* In this phase, each replica sends a COMMIT message to the other replicas. The replica then waits for at least $2f + 1$ COMMIT messages from other replicas (possibly including its own) that match the original PRE-PREPARE message received.

Once these three phases are completed, the replicas can execute the operation requested and send a REPLY message to the client, that waits for $f + 1$ replies to deliver the result, assuring that at least one correct replica executed the operation. Which, in turn, assures that at least $f + 1$ replicas committed the request. This means that PBFT, can make progress even when in the presence of faulty replicas. A faulty primary can slow down the system significantly, when the current primary is suspected to be faulty, a view change occurs and another replica is selected as primary. Figure 1 shows the operation of the algorithm in the normal case scenario.

The design of PBFT includes a number of choices that aim at improving the performance of the system. For instance, the protocol relies mostly on message authentication codes (MACs) based on symmetric keys, instead of using digital signatures and public key cryptography to authenticate the messages. This is because MACs are several degrees faster to create and to check than digital signatures. PBFT also tries to reduce communication costs, first by having only one replica send the full reply while the others only send an acknowledgment of it. Second, it proposes an optimized protocol to deal with read-only requests, where the replicas tentatively respond to the client, which then waits for $2f + 1$ replies (if there are conflicting writes operations the client needs to re-transmit the request as a standard read-write). Finally, in the last optimization, the replicas tentatively execute the client's request when the prepare phase is completed, as it is very likely that the commit phase will be successful. The client waits for $2f + 1$ tentative replies to deliver the result and, if there are conflicting replies,

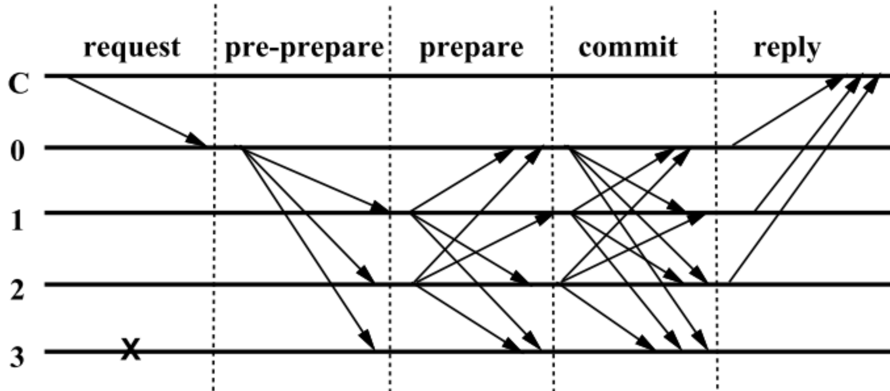

Fig. 1. PBFT Normal Case Operation; f=1, replica 3 is faulty. Taken from [2].

the client falls back to the normal protocol, resending the request and waiting for $f + 1$ "normal" replies.

3.1.2 Zyzzyva Zyzzyva[5] is a protocol that uses *speculation* to improve the protocol's performance in the best case scenario. As PBFT, Zyzzyva also requires at least $3f + 1$ replicas and also uses the notion of views, where one replica is selected to play the role of a primary. In Zyzzyva the role of assigning the order to the requests also falls on the primary replica. However, unlike PBFT, the replicas in Zyzzyva speculatively execute the request upon its reception, without running an expensive agreement protocol. This means that, in some cases, namely when faults occur, different replicas may (speculatively) execute different requests in different orders, and a conciliation algorithm needs to be executed in order to reconcile the replicas state.

In Zyzzyva, the client takes a more active role, and helps the replicas in assessing the success of the speculative steps. As in PBFT, the client sends the request to the primary, which assigns a sequence number to the request and then sends it to the backup replicas, establishing its order. If the primary is not faulty and is not suspected, all correct replicas will process these requests in the same order, and provide identical responses to the client. After collecting replies, the client considers the request completed when one of the following conditions is met:

1. If the client receives $3f + 1$ consistent responses from the replicas, it can consider that the request is complete. This is known as the *fast case*;
2. In the *two phase* case, the timeout set by the client upon sending the request is reached and the client still received between $2f+1$ and $3f$ consistent responses. In this case the client will provide help in terminating the protocol. For that purpose, it selects $2f + 1$ identical replies to create a *commit certificate* (this is proof that it has indeed received a sufficient amount of consistent replies) and sends it to all replicas, to ensure that correct replicas

become aware of the success of the speculative phase. Once $2f+1$ replicas acknowledge and reply to the client, it can consider the request completed and deliver the result to the application.

Figure 2 shows how the protocol works for the two scenarios described above. If the client detects valid responses with different sequence numbers for the same view, it as a *Proof of Misbehavior* of the primary, that when sent to the replicas triggers a view change. In another case, if the client does not receive at least $2f+1$ identical speculative replies it will conclude that the speculative execution has failed. Zyzzyva fallsback to a protocol that is similar to PBFT: it retransmits the request to all replicas which, in turn, resend it to the primary. The replicas set a timer, if no progress is made and enough replicas consider the primary faulty, a view change occurs, ensuring that the protocol will eventually make progress on the requests. If the replicas receive the request from the primary, they once again speculatively execute it and respond to the client. This is where Zyzzyva under performs in relation to PBFT, if the primary is faulty, the speculative work done is wasted and needs to be redone in a different view, yielding a worst case scenario.

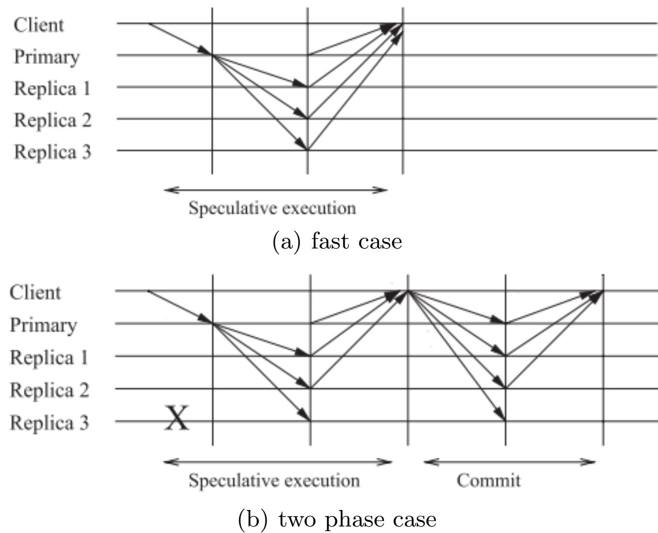

Fig. 2. Zyzzyva Message Patterns; f=1. Taken from [5].

The implementation of Zyzzyva uses a number of additional optimizations. Like PBFT, Zyzzyva uses of MACs and authenticators instead digital signatures for most of the messages in the system in order to reduce its computational overhead. Like PBFT, Zyzzyva also has a optimized protocol for read-only requests. As a way to reduce the bandwidth consumption, when sending replies to the

client, only one replica responds with the full reply while the others send only digests of the response.

3.1.3 Quorum and Chain The works above show that it is possible to derive different BFT protocols, that aim at different operational conditions. In particular, Zyzzyva can outperform PBFT in fault-free runs (which should be most common anyway). These ideas were explored in [7] to develop a system, called Aliph that can run multiple protocols, for different operational conditions. To switch among these protocols, the authors propose an abstraction that all protocols should implement, called *abortability*, that captures the required properties to support dynamic reconfiguration. The underlying idea of Aliph is that it should be possible to abort the execution of a protocol that can't perform under the current system's conditions, and replace it with another more suitable protocol.

With this framework in mind, the authors take the ideas of Zyzzyva even further, by proposing protocols that work under very narrow conditions, but that can be easily replaced. In this context, the authors propose two protocols, namely Quorum and Chain, that we briefly describe below.

Quorum is variant of the speculative version of Zyzzyva that is optimized for the case where there are no faults and there is a single client proposing commands, meaning no contention (and, therefore, replicas cannot receive different commands in different orders). In this protocol, the client sends the request directly to all replicas, that execute the request speculatively, and waits for $3f+1$ identical responses. If speculation works, the client can commit the request. If $3f+1$ responses are not received or if some responses do not match, the protocol is simply aborted. Since Quorum responds to the client with a one round-trip it achieves very low latency, even if only for a very specific scenario. The message pattern for this protocol in depicted in Figure 3(a).

Chain is a protocol that, like Quorum, is optimized for the fault-free case but can handle concurrent requests from multiple clients. The protocol is a BFT variant of the Chain-Replication[12] protocol. The Chain protocol structures the replicas as a pipeline, where all of them know the fixed ordering of replicas. The first replica in the order is the *head* of the chain and the last is the *tail*. The client sends its requests to the head of the chain, that in turn assigns a sequence number to the request, functioning like a primary. After assigning the sequence number it sends the request to its successor and so on, until it reaches the tail which responds to the client, as shown in Figure 3(b). Each replica only accepts messages from its direct predecessor, with the exception of the head. A technique called chain authenticators(CA's) allows to provide a proof to the client that the request was processed correctly by the replicas. This protocol can offer good throughput in absence of failures, due to its "pipeline" form of processing, but requires an expensive reconfiguration of the chain if faults occur. To avoid such costs, in case of failures, the protocol can simply be aborted and replaced by another protocol.

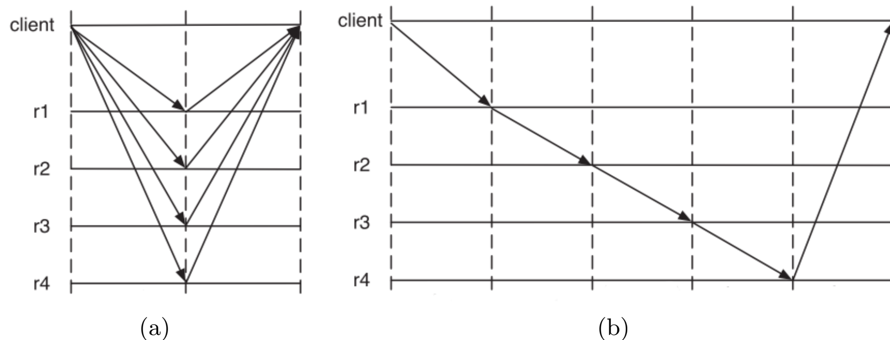

Fig. 3. Quorum (a) and Chain (b) Message Pattern; f=1. Taken from [7].

3.1.4 Discussion We chose these protocols because we believe they exemplify how protocols change in function of what they are built to do, namely how the common case they consider impacts how the protocol works and the assumptions they make.

PBFT was built to handle contention and still perform consistently under failures and asynchrony as its common case scenario assumes these scenarios. Zyzzyva, on the other hand, assumes that faults are not common and tries to gain performance by not running an expensive agreement step, risking using a costly recovery scheme if things go wrong. The last two we discussed only run for very specific contexts, deferring to other protocols when that context is no longer favorable according to their running conditions.

From analyzing these protocols we can see that a one-size-fits-all protocol does not exist, and that, to the best of our knowledge,no protocol outperforms the others in every possible scenario.

4 Adaptive Systems

Nowadays, systems tend to run in highly variable contexts, where conditions frequently change, e.g. variations to network latency and bandwidth, to the number of clients and more. In these kind of environments, a monolithic system's performance will likely suffer as it can only perform for a restrict set of conditions for which it was developed. Adaptive systems, on the other hand, are able to react to said events by changing their configuration and/or structure, e.g. adjusting resource bounds, triggering machine relocation or even switching the underlying protocol. Since these adaptations can be defined for a wide range of scenarios, adaptive systems can be very flexible and embrace those events, allowing them to maintain or even improve the quality of the service.

An *Adaptive System* is then a system capable of altering its configuration during execution in response to changes to its context in order to achieve or approach a defined goal. The operation of an adaptive system usually involves the collection of data regarding the current performance of the system. This data

is used to compute a number of metrics that can be subsequently used to reason about the need to perform adaptations. These metrics feed an evaluation process, namely an adaptation engine, that will check if the current configuration is still appropriate for the status of the environment or still in line with the system goals. If it is no longer the case, an adaptation may be triggered and changes will be made to the system.

An adaptation policy defines in which conditions the system should be adapted and what adaptation should be applied in each scenario. There are many different ways to express policies, from high level approaches[13, 14] (that mainly specify the system goals and let the choice of the required adaptations be performed automatically from those goals) to low level event-condition-action rules[15], that specify exactly which adaptations should be performed in response to each event. In any case, the policies embody, explicitly or implicitly, a set of business goals that the system should strive to achieve. Policies can be defined before the system is deployed or can be created or improved during execution[16, 9].

The architecture of an adaptive system typically allows the implementation of a control loop, known as the *MAPE-K* loop[17], whose name is derived from its main components, namely the Monitor, Analyze, Plan, Execute, Knowledge. This control loop captures the main responsibilities and concerns that occur in an adaptive system. Thus, an adaptive system can be modeled as set of sub-systems and components that interact among each other, each assigned with some of the responsibilities mentioned above, as illustrated in Figure 4. When deployed, the five components of the control loop and the system being managed, can be clustered in three different sub-systems, namely:

- *Monitoring system*, the system that will collect the metrics from the system context by interacting with its "external" components, the sensors. The sensors collect relevant metrics, for the adaptations, from the managed system's context. This sub-system takes care of the Monitor and part of the Analysis concerns of the system and is the target of this report;
- *Adaptation System* receives the metrics collected by the Monitoring system and evaluates the state of the managed system against its goals, triggering adaptations or reconfigurations if necessary. This component will do its own analysis of the metrics and then plan and execute policies in the managed system, adapting it to the current context;
- *Managed System*, the actual provider of services to the clients and the one that needs to be able to resist the changes of its context to continue to provide good quality of service to its users. The others depend on the type of this system, namely for policy definition and metric collection.

4.1 State Machine Replication

In this section we will discuss the Replicated State Machine System, the managed system that we want to monitor with our work, and analyze its characteristics and capabilities, more concretely what can be adapted in such a system.

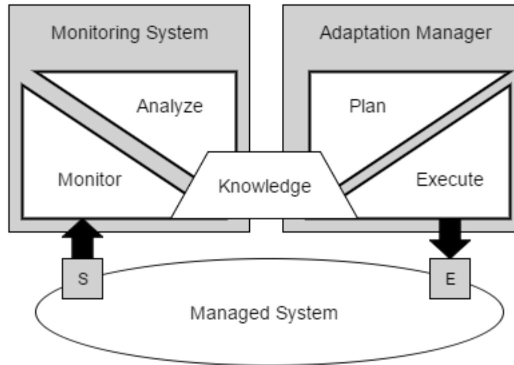

Fig. 4. Generic Adaptive System Architecture.

State Machine Replication(SMR) is a technique to build robust services by taking advantage of active service replication and state propagation. Thanks to it, the services built on top of a replicated state machine are able to tolerate faults and keep providing service to the clients even in their presence. The type of faults it can tolerate depend on the underlying protocol of the state machine, we will go into more detail about it ahead.

An SMR system works by running several copies of a service, simultaneously, usually one per machine to take advantage of different fault probabilities. This system is composed by three sub-algorithms, namely agreement, view change and checkpointing. The agreement was discussed in Section 3.1 and view-change is the process by which the primary in the agreement protocol is changed, as such, view-changes are also usually handled by the protocol. In terms of checkpointing, each replica keeps a log of its activity, recording the different agreement processes, operations executed and responses given to the clients, effectively working as the replica's state. After a certain amount of entries are added to the log, the checkpointing process is triggered and the log is trimmed, thus creating a checkpoint. A checkpoint represents all the entries added before its creation, and as such these entries can be safely deleted from the log, keeping it from growing indefinitely. If a replica lags behind or a new one is introduced they can request a state transfer from the other replicas that, in turn, send their checkpoints and "uncheckpointed" log entries effectively updating/recovering the requesting replica's state.

Analyzing this type of system and looking at an actual implementation, namely *BFT-SMaRT*[10] an open-source SMR library whose adaptive implementation is the concrete monitoring target of our work, allows us to get some insight into what can be adapted for this type of system. From our review of the literature and implementation we denote four main types of adaptations that may occur in an SMR system:

- SMR's parameter reconfiguration;
- Reconfiguration of the SMR's fault model;

– Protocol Switching;
– Adaptations targeting the system's replicas;

4.1.1 SMR's Parameter Reconfiguration Several parameters are set when a SMR system is deployed, as such, it stands to reason that some of said parameters could also be reconfigured to accommodate for changes in the system's environment.

While the system is occupied with a previous request, incoming requests are accumulated in message queues. These requests can then be batched for the agreement process allowing multiple operations to be executed with a single agreement, decreasing the cost of running said agreement. The size of the system's message queues and batch can be a reconfigurable parameter in the system. E.g. If the latency in the system is increasing, increasing the size of the batch will reduce the number of agreements, allowing the system to cope with the increase in latency. Or if the amount of requests keeps increasing and the system still has free memory, the size of the message queues can be increased along with the size of the batch to handle that load, instead of just dropping requests.

The messages exchanged within the system utilize some form of cryptography to ensure its authenticity and integrity. The two main techniques used in an SMR system are digital signatures and MACs, each with its own strengths and weaknesses. As such, we could reconfigure this property as necessary, either to gain performance (using MACs) or a stronger authenticity premise to the messages, like non-repudiation.

The last parameter for reconfiguration we will discuss is the value to trigger the checkpointing process. As shown in [15] this parameter, for the BFT-SMaRT library, has different ideal values depending on the number of replicas in the system. If an adaptation to scale the replication factor either up or down exists for the system, then checkpointing threshold should also be reconfigured.

4.1.2 Fault Model Reconfiguration SMR systems can also be configured in terms of their fault model, more specifically, they can run one of two types of fault tolerance, *Crash Fault Tolerance*(CFT) or *Byzantine Fault Tolerance*(BFT) for the agreement step. This configuration will have an impact on other possible configuration parameters, namely the number of servers necessary to tolerate f faults thus changing the replication factor, $2f+1$ and $3f+1$ for CFT and BFT respectively, and the underlying protocol used in the agreement step.

Considering certain business models where, for some periods of time, the system's throughput becomes more important and some amount of "loss" is acceptable, an interesting approach would be to allow the system to switch its fault model from BFT to CFT, by changing the underlying protocol, e.g. switching some protocol like Zyzzyva or PBFT to something like Paxos[18], also effectively increasing the amount of crash faults tolerated. By running the less expensive fault model, the system would be able to increase its performance and be able to handle more requests thus possibly still compensating for the loss in

robustness. The opposite is also possible, changing from CFT to BFT if some erratic behavior is detected, gaining robustness at the cost of performance.

4.1.3 Protocol Reconfiguration

As discussed in Section 3.1, there are a wide range of different protocols but none that outperforms the others in every scenario. As such, a meaningful adaptation that can be done in an SMR system is switching the underlying protocol to suit the current context of the system, as evidenced in [7, 9]. With this adaptation we can create a system that can leverage the benefits of existing solutions and mitigate their shortcomings.

Protocol switching can be done in one of two ways, a static approach or a dynamic one. In the static approach there is a fixed order for the protocols, when the conditions do not suit the current protocol, it aborts and a switch to the next one in line is made, as shown in Aliph[7]. As is done in Aliph, we could have an order like Quorum to Chain and to Backup (authors use PBFT). This approach suffers by not taking advantage of other information about the characteristics and performance of the protocols. This is where dynamic switching comes in, by analyzing how the protocols behave in a wide range of scenarios we can make more educated switches as evidenced in ADAPT[9]. Even if the context of the system is still in line with the protocol in place, another protocol can have better performance in the same context, as such a switch to this second protocol would be beneficial to the system.

4.1.4 Replica Adaptations

Since an SMR system uses a replication factor to mask faults, we could introduce adaptations at this level. The replication factor could be reconfigured as threat level changes by scaling up or down the number of replicas in the system[15] or even by moving replicas to other machines in a more “secure” location.

Also, considering that the system can only make guaranties about the safety and integrity of its service if the number of faulty replicas does not exceed f, it would also be reasonable to have adaptations that deal with faulty replicas, more specifically having the capability of:

- Reconfiguring a faulty replica;
- Switching the replica out for a “fresh” one;
- Moving the replica to another machine;

The adaptation of moving replicas from one machine to another can also have another purpose. It can be done as a way to increase the capability or performance of said replica by increasing the base resources it has access to, because some co-located workload is incompatible with the system’s workload[19, 20] or even if the probability of failure for that machine has increased[21].

5 State Machine Replication System Monitoring

In this Section we will discuss the *Monitoring System* which is the main focus of this work. Namely, we will look at the general architecture of the monitoring

system, what kind of metrics will be collected from the monitored system, and what type of faults the monitor system will detect.

5.1 Monitoring System's Architecture

One of the main purposes of the monitoring system is to collect metrics about the performance of the monitored system and about its operational environment. The collection of metrics is done by deploying an infrastructure of components called sensors. The sensors are deployed along side the managed system, having the responsibility of monitoring it, by reading metric values and detecting relevant events that may occur. Sensors can be placed in the processing nodes, to measure metrics such as CPU utilization, memory utilization, etc, or in networking components, to measure metrics such as packet loss, available bandwidth, etc.

Values collected by the sensors are typically sent to a logically centralized component of the monitoring system. This component, that we will simply name the monitoring broker, is in charge of aggregating and filtering data collected by the sensors (for instance, by masking erroneous values sent by faulty sensors) and in charge of sending the processed sensor information to the Adaptation Manager (AM). In turn, the AM will use the values provided by the monitoring system to feed the policies that define how the system should be adapted. The transfer of information from the sensors to the broker can be performed by pushing the information from the sensors to the broker, by having the broker periodically pull the information from the sensors, or by any combination of these two approaches.

5.1.1 Fault Tolerant Monitoring System As with any other system component, a sensor may be subject to faults. A faulty sensor may provide inaccurate readings, that do not reflect the real state of the system. If the faulty reading escalates to the Adaptation Manager and the policies are not robust enough to tolerate such inaccurate readings, one may induce the system to perform non-optimal adaptations, or even adaptations that may cause the system to fail. Therefore, whenever possible, we would like the monitoring system to implement mechanisms to filter inaccurate values that may be potentially provided by faulty sensors. In this way, at least for some metrics, the monitoring system can guarantee the delivery of accurate information to the the adaptation manager, simplifying the specification of the adaptation policies. Typically, tolerance to faulty sensors can be achieved by replicating the sensors and then by applying some voting function to the values provided by different sensors. Note that, in some cases, it is not feasible to install redundant sensors, and the faulty values must be handled explicitly at the AM level. For instance, if a node has been compromised by an attacker, it may be impossible (but most likely, also irrelevant) to determine exact value of CPU utilization at that node.

Naturally, the monitoring broker (that collects the values from the replicated sensors and applies the filtering functions) may also be subject to faults. Therefore, the broker itself needs to be replicated. Since that, in turn, the adaptation

manager will also need to be replicated, in the report we advocate a deployment where each replica of the monitoring broker is colocated with each replica of the adaptation manager, such that the fault-tolerance of these two components can be dealt in an integrated form. However, the need to replicated the monitoring broker opens the door for a faulty sensor to send conflicting values to different replicas of the broker. In turn, this may cause different replicas of the broker to pass conflicting monitoring information to the different replicas of the adaptation manager. To avoid this issue, some form of agreement must be executed among replicas of the monitoring monitor.

A potential problem that may occur when building a fully fault-tolerant monitoring system is that the need to use replicated sensors may induce a significant load on the monitoring tasks. Assume that each sensors is replicated using $3f+1$ replicas and that the system requires the use of x sensors, we may need to deploy $x(3f+1)$ sensors, which can be an high number even for small values of f and x. For instance, just to tolerate 1 faulty sensor, and by having 5 different types of sensors, we would need to deploy 20 sensors' replicas in the system, each sending readings to the different replicas of the broker, which would account for 80 messages for each monitoring cycle. To mitigate this problem we will support different fault models and different replication degrees for each class of sensors, such that information regarding the semantics and the implementation of sensors can be used to avoid unnecessary redundancy. For instance, it may happen that the implementation of a given sensor makes the occurrence of byzantine faults unlikely, or that, for some metrics, the adaptation policy is robust to inaccurate readings, and therefore less than $3f+1$ replicas need to be used.

Another potential problem is the amount of coordination that needs to be performed among replicas of the monitoring broker to ensure that consistent outputs are generated. To run a separate consensus for each value received from a (potentially faulty) sensor is certainly prohibitively expensive. Therefore, we aim at batching agreements of individual readings, and performing these multiple consensus in parallel.

5.2 Relevant Metrics to Monitor

The exact sensors to be implemented during this project are still unknown at this stage. This will naturally depend of the properties of the managed system, the types of adaptations it supports, and on the needs of the adaptation policies that are going to be implemented by the adaptation manager. These components are also still under development at this stage. Nevertheless, in this section, we aim at identifying some of the relevant metrics that may be important to capture by the monitoring system that we aim at developing.

The main concerns with a SMR system are its performance and robustness. This performance and robustness can be affected by a number of factors directly or indirectly. As such, we identify some of these impact factors and provide some examples of the respective metrics that the monitoring system needs to capture:

- *Workload Patterns:* The number of request per unit of time that are made to the system, and the characterization of these requests, has an obvious impact

on the system performance. Besides the presence of faults, the impact factors that most significantly affect the protocol's performance are number of clients, request size and response size[9]. In most cases, the workload changes with time, such that it may be relevant to adapt the system configuration accordingly. Information regarding the workload can be measure at the system interface, for instance measuring the number of requests that arrive to the proxies, or indirectly though the observation of the state or performance of some system components (for instance, inferred from the size of request queues);

- *Network Utilization:* In a distributed system, the network is often a bottleneck that can limit the system performance. Different protocols make different tradeoffs between network utilization and other properties, such as latency. For instance, quite often it is possible to reduce the latency of a protocol by sending more messages; naturally such latency benefits are only experienced if the network is not saturated. Information about the network utilization, as well as information regarding other network properties, such packet drop rates, can be useful to select the right system adaptation;
- *Machine Utilization:* Other potential bottleneck source in a distributed systems are the resources of each individual node, such as memory and CPU. Elastic scaling, the ability to automatically increase or decrease the number of servers in response to changes in the workload, is a classical example of a type system adaptation that can benefit from detailed information regarding the resource utilization at each node. Furthermore, as we have seen, BFT protocols are often asymmetric, and there is often one node (the leader) that has more tasks than the remaining nodes (the backups). Knowledge about the resource utilization of each machine in a BFT configuration may be useful to select the most appropriate node to execute the role of the primary in a failure-free run.
- *Threat Level:* Byzantine faults are often caused by a malicious agent that is able to intrude the system. A number of intrusion detection mechanisms exist that are based on detecting anomalous patterns in the system behavior that may indicate attempts by an adversary to intrude the system. Such information can be used, with information from other sources, to compute a threat level that is correlated with the likelihood of malicious faults and that may be used to reconfigure the system to a configuration that is more robust, even if the cost of a performance loss (by using more replicas or by using more diverse, but less optimized, implementations).

5.3 Fault Detection

The monitoring system can also detect and report the occurrence of faults. Knowledge about the faults and their location can be used by the adaptive system to automatically perform corrective measures, for instance by re-starting replicas or launching new replicas to replace failed replicas of a given component. Note that even if the managed system is able to mask a fault, this information is critical to allow the system to recover the original degree of fault-tolerance.

For instance, a typical BFT protocol can continue to operate if one of the replicas is faulty, but the faulty replica should be replaced before new faults occur. Also, an early detection of faults, and a quick triggering of corrective measures, may ensure the containment of errors to a small subset of components, making recovery more efficient.

There are two approaches to *fault handling*:

- *Fault Masking:* We discussed this approach in Section 3, it utilizes a replication factor to hide the presence of faults to the clients up to a certain number of faults.
- *Fault Detection:* This method to fault handling consists in detecting faulty components so that they can be either removed or repaired. This approach will the topic discussed in this section.

5.3.1 Background The work done in the context of *Fault Detection* is extensive, specially in attempts to solve the consensus problem in the presence of faults, evidenced by the work done by Chandra et al. [22] where they introduced the concept of unreliable failure detectors and study their potential to solve the consensus in the presence crash faults, which is later expanded to byzantine faults by Kihlstrom et al.[23] and Malkhi et al.[24], to name a few.

Unreliable Failure Detectors have been categorized as having two properties[22]:

- *Completeness:* There is a time after which every process that crashes is permanently suspected by some correct process.
- *Accuracy:* There is a time after which some correct process is never suspected by any correct process.

The definition of these two properties will depend of the strength of the detectors considered as well as the faults they detect, the one presented above is in relation to the weakest failure detector considered in Chandra et al. work in the context of consensus in the presence crash faults. E.g Kihlstrom et al. and Malkhi et al. present these properties in the context of byzantine faults, or Doudou et al.[25] that present them for a more particular\restricted class of byzantine faults, namely muteness faults.

Failure detectors are not omnipotent, and as such there are limitations to what can be detected, such that byzantine faults can be divided into *non-detectable* and *detectable* failures[23].

- *Non-detectable Byzantine faults* are faults that cannot be detected from received messages or that cannot be attributed to a particular process, e.g a Byzantine process that sends a different initial value than the one it was supposed to.
- *Detectable Byzantine faults* can be defined as two different sets, *omission* faults and *commission* faults. *Omission* faults occurs when a process does not send a required message to all necessary processes.

Commission faults, on the other hand, are divided into two types, processes that send messages not properly formed or authenticated and processes that send divergent messages to the different replicas.

The Crash Faults can be defined as a subset of byzantine omission faults, as such we focused on the classification of the byzantine fault type. In Figure 5 we can see a visual representation of this fault classification.

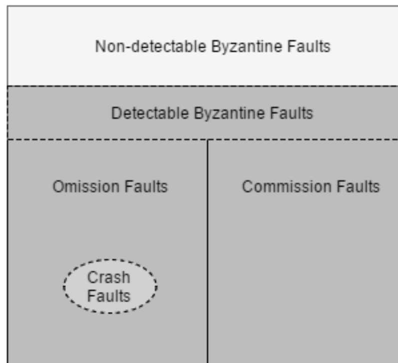

Fig. 5. Byzantine fault classification.

Considering that the presence of these faults can also be because of an attack perpetrated with malicious intent and not simply an arbitrary occurrence, another interesting concept to employ in fault detection is *Intrusion Detection*[26]. This type of detection works by scanning the network or machines for signs of "foul-play" that could indicate that an attack is occurring, a machine has been intruded or that such attempts were made. E.g by detecting an unusual amount of port scans done to the machines where the replicas are located or detecting a great amount of requests with the purpose of taking the system offline. However they also have some issues, they are either based on heuristics that can produce an high quantity of false positives and negatives or require a formal specification of what to expect out of a system, which for complex systems, like one that utilizes protocol switching, can be hard to produce or maintain.

5.3.2 Interest in Fault Detection Haerberlen et al.[27] present a case advocating in favor of fault detection, and very concisely expose the main advantages of this approach, such as enabling a timely response to faults detected and serving as a deterrent to attacking the system itself, among others.

Although this work is directed at monitoring a system that utilizes fault masking, both approaches are complementary to one another. Fault detection can be used to enhance the resilience of the masking method to fault handling. Since a system employing the fault masking technique can only mask the faults to the clients and guarantee the service correctness up to a certain number of

faulty replicas, its useful to be able to detect which of the replicas of the system are the faulty ones and repair them preventing the number of faulty replicas from surpassing that threshold, thus extending the system's life and preventing severe harm from occurring.

Some systems that utilize the concept of fault detection have already been introduced:

Falcon[28] is a failure detector that leverages internal knowledge from various system layers to present sub-second crash detection and reliability with little disruption. Falcon uses a network of spies that monitor the various layers and report if they are UP or DOWN. Considering that there is some degree of possibility for a false positive where a reported DOWN layer can start sending messages, Falcon's spies as a last resort kill the layer they suspect to be DOWN, making that report definitive, aiming at the smallest layer possible. This killing process introduces a degree of cost to deal with false positives as it can be coarse-grained or fail under the presence of network partitions. In response to this, Albatross[29] was introduced as an improvement over Falcon. This failure detector still utilizes the spy network introduced with the previous system, but instead of killing the layer to create a definitive report it uses Software Defined Networks, modifying it to prevent the messages of the suspected crashed processes to get through to the correct ones.

PeerReview[30] is another failure detection system but for a byzantine context. Its aim is to provide accountability in distributed systems in a general and practical way, ensuring that byzantine faults observed by correct nodes are eventually discovered and linked with the faulty node. In PeerReview each node keeps a tamper evident log that records the messages sent and received by it as well as the input and outputs of the application, this allows the detection of deviant behavior by a particular node. The system thus requires that the messages are signed in order to guarantee their non-repudiation and prevent spoofing, allowing their utilization as proof to show faulty behavior.

The last example we will discuss is ByzID[31], a Byzantine fault-tolerant protocol that tries to approach the costs of crash tolerant algorithms by utilizing an Intrusion Detection System (IDS). This protocol relies on a specification-based IDS to detect and suppress primary equivocation, enforce fairness, detect various other replica failures and trigger replica reconfiguration, making it almost like a Primary Backup protocol that tolerates byzantine faults, requiring only $2f+1$ replicas. It is able to this by integrating an instance of said IDS into each replica to monitor and discard messages that are not in conform with the specification. To have this kind of responsibility and power, the IDS needs to be built on top of trusted hardware to become a trusted component for the system and, although the authors claim that this IDS is simple and lightweight in order to be able to be built as a trusted component, we believe that in practice this might be a more complicated assumption. Considering the power the IDS has in order to keep the replicas in check, it might not be as lightweight as assumed, and no practical implementation in trusted hardware is presented.

6 Architecture

In this section we propose the architecture for the target of this paper, the *Monitoring System* (MonS).

6.1 Overview

As discussed in the related work above, the MonS is one of three main components in an *adaptive system*, the monitor will have to communicate with these other components, either to monitor the components, in the case of the *Managed System* (MS), or to relay the information collected in the case of *Adaptation Manager* (AM).

The MonS consists of a monitoring broker which we will designate as the *Aggregator* and an infrastructure of smaller components, the sensors. The sensors will collect data from that MS's context and from the MS itself and then transmit that information back to the aggregator. The sensors are then, the points of contact between the MS and the MonS. The aggregator takes that data, processes it and then feeds it to the AM, thus acting as a proxy for this information and being the point of contact with the AM. The interactions between the components of the MonS and between the other subsystems of the adaptive system are exemplified in Figure 6

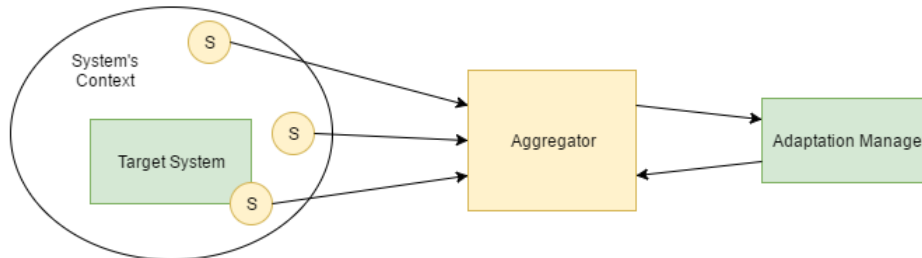

Fig. 6. *Monitoring System* interactions in the adaptive system. Components in yellow are part of the *Monitoring System*, the ones represented by the S are the Sensors.

As discussed in section 5.1.1 the MonS will have to have a fault tolerant architecture. The aggregator will be built using the BFT-SMaRT library as a basis, but with some alterations, considering responsibilities of this component that will be discussed in section 6.3. The sensors will be built as simply as possible to limit the overhead their presence will have on the MS. They will merely collect the metric values and send them to the aggregator. As stated in section 5.1.1 in order to achieve this simplicity, even for the replicated sensors, the responsibility of running the consensus for the values collected will be pushed onto the aggregator. The aggregator will have to receive the different values for the metrics and aggregate them as it will be discussed in section 6.3.

6.2 Sensors

We divide the sensors into three types depending on the type of faults they tolerate, namely non-replicated, crash tolerant or byzantine tolerant. This distinction needs to be made because the conditions for each sensor change according to where they are deployed or the type of metric they gather, and some of their configuration changes as a result. Each sensor (if singular) or sensor's replicas (if replicated) are defined by four configuration parameters. (1) The identifier, a value that identifies the sensor or sensor replicas and the metric that it collects. This is used to identify the values obtained for said metric before the AM; (2) their type that as referred above we separate into three different categories; (3) IP Address to identify the sensor in the network; and (4) a public key (or keys) used for creating secure communication channels or authenticate the messages transmitted by it.

Considering the impact the sensors can have on the MS's environment and its performance, as we referred before, they being as simple as possible is a key requirement to keep in mind for their implementation. The type of metric that the sensor will target and the place where it will be deployed will influence how the sensor will be implemented and the process by which it will collect the metric. As such, some considerations need to be made for their implementation like operating system, programming language, available libraries that depend on the previous two, among others. Some sensors will possibly require that the MS implements an API in order to collect accurate information about its internal state, e.g sensors that track the correctness of the messages structure, as the MS will adapt its protocol, message structure might also change.

As discussed in sections 4.1.1 and 5.2 the metrics collected reflect important factors that impact the MS. We divide these metrics into three categories:

- *Performance Metrics:* Data that directly affects or is related to the performance of the MS, such that its collection can give insight on how the system is performing. This includes metrics such as latency, throughput, etc;
- *Resource Metrics:* As the name implies this applies to information on how resources are being spent and their availability in the MS, e.g CPU consumption or available memory;
- *Fault Metrics:* Metrics that indicate the presence of faulty components or threat level in the overall system.

Although some metrics can be categorized as belonging to more than one type, we consider their primary application for this categorization. In Table 1 we present possible sensors considered to be integrated into the MonS, divided with the categories presented above.

6.3 Aggregator

In this section we will explain in detail how the aggregator is constructed, how it works and its features. In Figure 7 we see how the aggregator is composed.

Table 1. Sensor Examples.

(a) Performance Metric Sensors.

Sensor	Description
Latency	Records the latency present in the system
Throughput	Measures the amount of requests being processed by the system
Capacity	Collects the number of active clients sending requests to the system
Request & Response Size	Captures the size of incoming request and respective responses

(b) Resource Metric Sensors.

Sensor	Description
CPU Consumption	Records the CPU utilization percentage for the replicas machine
Disk I/O	Collect disk I/O statistics, namely reads and writes
Memory Consumption	Captures the memory consumption percentage

(c) Fault Metric Sensors.

Sensor	Description
Quorum	Records the replicas that participate in the quorum of the agreement Protocol
Crash	Checks if the replica's process is still running
Muteness	Checks to see if a replica is not sending the messages that it should
Divergence	Checks if the messages that a replica sends to other replicas diverge
Incorrect Messages	Checks to see if the messages are properly formed and authenticated

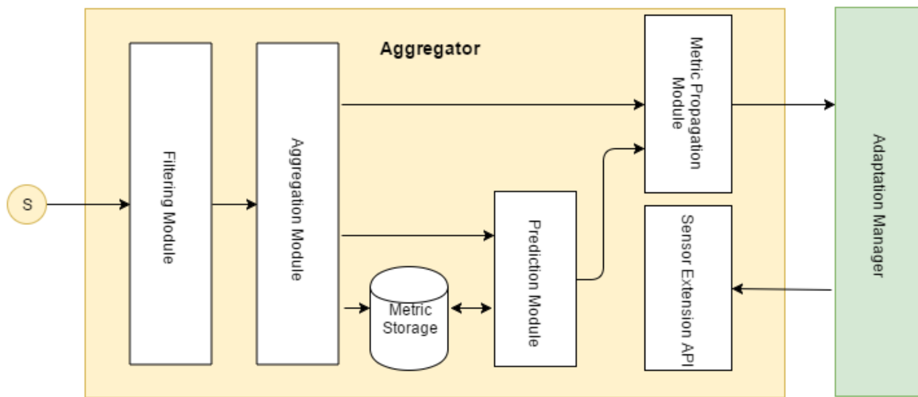

Fig. 7. *Aggregator* internal workings. (component S is a Sensor).

6.3.1 Aggregation Function

In order to save on the cost of adding *fault tolerance* to the sensors and help keep them as simple as possible, the aggregator is in charge of the data condensation, namely each set of replicated sensors will have an aggregation function associated with it, e.g some basic operation like averaging the result, discovering the median of the values received or even a composition of operations to create a more complex aggregation. This function is applied to "rounds" of metrics as they are collected (the sensors gather metrics sequentially, sending the metrics with its attached sequence number, thus composing a "round"). The result of the aggregation function is then sent to the AM in a robust manner. Another application of this function, besides aggregating the replicated metrics, is applying some degree of processing to the data, such as transforming the raw information into to more easy to use data for the AM, thus serving some purpose even for non-replicated sensors. This function also needs to be deterministic in order to guarantee that every replica arrives at the same result.

The aggregation function by itself is not enough to guarantee the integrity and equality of its result across the different replicas of the aggregator, e.g if one of the sensor's replicas is faulty it could send different values to each replica of the aggregator, effectively making each of them output a different value when aggregating the results. As such, before applying the aggregation function the diverging values for the replicas of the sensors need to be filtered out.

In order to achieve this we need an extra filtering step before the aggregation function is applied to the values received. This is done by running a consensus when the metrics are received ensuring that by the end of it, all correct replicas of the aggregator will have the same values as input for the aggregation function.

6.3.2 Prediction Function

Since the adaptations executed by the AM are triggered by changes in the state of the environment, if we could somehow understand how the context will evolve we could have a system that preemptively triggers such adaptations in response to events before they even happen. To accomplish this, we need to look at current and past events that occur in the MS's context and try to predict what might be coming ahead.

This function is defined and then associated with the sensor much like how it is done with the aggregation function, but unlike it, the prediction function also takes into account past events, as such metric values for sensors that have this option defined will have to be stored for future use. The final result of this function will be a tendency, so by receiving the input and crunching the data it will output a supposition on how the values of that sensor will develop in the future. If they will stable, have a tendency to rise or decrease, e.g if the latency values have been steadily increasing for the past twenty collections made, then a very simple prediction function could look at this and say that latency as a tendency to continue to go up. Also, considering how some factors impact each other, this function could also take into account the data from other sensors if available.

6.3.3 Metric Propagation

When the final value for the metric is obtained, the aggregator replicas will have to propagate it to the AM. To achieve this we have two options. In the first one we treat each of the aggregator's replicas as a process that shares the same machine with a replica of the AM, and as such the value can be passed directly from the aggregator to the AM. This works because all correct replicas of the aggregator possess the same result for the metric collected, and even if some replicas of the aggregator are faulty since they are co-located with the processes of the AM, those replicas of the AM will also be faulty. The main advantage of this method is the resources saved in the message exchange, each process sends/receives exactly one message. On the other hand, the MonS and AM become integrated with one another, thus imposing some restrains to the AM, namely it terms of its replicas deployment. Since the replicas will have to be co-located, we cannot take advantage of different machines to power both the system monitoring and adaptation engine separately.

The second approach would be to have each replica of the aggregator propagate its result to each replica of the AM. When the replicas of the AM receive $2f+1$ messages with the same value, they can deliver it to the policies for consideration. The amount of messages sent with this approach is its main drawback, since we would have each replica of the aggregator send a message to each replica of the AM, thus having a great number of messages circulating in the system. Although this effect can be mitigated with the use of batching and sending various results to the AM, it will still tax the system. Unlike the other approach, this does not have limitations in terms of replica deployment, as the MonS and AM are independent systems.

6.3.4 Extensibility

The MS's goals may be subject to change, introducing new needs and consequently new possible adaptations. This introduction may imply a need to gather new metrics from the execution environment, meaning the sensor infrastructure would also need to be expanded further than for what it was deployed, in order to accommodate this need. Furthermore, some adaptations, e.g replica relocation, as they are triggered, may shutdown or redeploy certain sensors as a result of it, possibly changing configuration parameters of said sensors. Considering this, there is a clear demand for the MonS to be built using a modular and extensible approach such that it could be augmented with new sensors, be able to disconnect retired ones, add and remove sensors' replicas or even seamlessly change the underlying aggregation or prediction functions of sensors.

The management of sensors needs to be made at the aggregator level, working with coordination from the AM. As such, this component will provide an API in order to receive requests to change these settings. This API will have to be robust considering the fault tolerance aspect of the system we are constructing. And so, considering that the AM is also a replicated component, in order for it to accept and execute the reconfigurations, the MonS will need to receive, going by the standard, $2f + 1$ equal requests from the AM replicas in order to guarantee the legitimacy of the demand.

To add a new sensor this API will need receive a set of parameters, some of the ones defined in Section 6.2 and some others, which will represent the sensor in the aggregator:

– *Identifier:* This will identify the sensor and the metrics it collects before the aggregator and the AM. As such, the most probable identifier would be the name of metric collected by it;
– *Type:* Indicates the type of sensor, either non-replicated, crash or byzantine tolerant;
– *Number of Faults:* This represents the number of faults tolerated by the sensor (0 if non-replicated);
– *IP List:* List of IP's for the sensor, one or more depending on its type. If the sensor is replicated. This will also define the order for the replicas;
– *Public key(PK) List:* List of PK's that will be used to establish a secure connection with the sensor in the network. Their order must be that of the previous parameter;
– *Aggregation Function:* This is the function used to aggregate the results. This is defined by implementing an interface, with a method that receives an array of inputs and outputs a result;
– *Prediction Function:* This is the function that will analyze the present and past data received from the sensor and try to predict how the system's context will evolve for that particular metric. By defining this, the values will be stored up to a certain amount also defined by this function. This function will also have to specify if it depends on other sensor's information, if the sensor does not exist or has not been associated with aggregator this will be kept in a dormant state.

To add and remove replicas of a sensor is just a case of sending the Id of the sensor, the IP or IPs of the replicas and their respective PKs. Depending on the case, its removed or added to the list of accepted inputs. When adding, the replica's value will be put in the end for the order in the array, and in case of removal the order is adjusted accordingly. To alter the aggregation function or any other parameter (except the identifier) is just a question of sending the parameter's value and the sensor's identifier using the provided API.

7 Evaluation

In this Section we define the metrics that will be used to evaluate our solution. We are interested in measuring the overhead the *Managed System* monitoring will have in its performance, how our solution performs and how accurate is our fault detection.

To evaluate how much overhead the *Monitoring System* causes, we will start by evaluating the performance (this will be measured in terms of throughput) of the *managed system* without any monitoring to establish a baseline. After this is established we will measure the performance with varying degrees of metrics

collected, by increasing the number of active sensors, and do a comparison of the results.

Another important metric to evaluate in how well the solution performs. We will evaluate the solution in terms of metric throughput towards the *Adaptation Manager*, with varying amounts of sensors active and in the presence of faults. Also important is how well the Filtering/Aggregation approach, described in Section 6.3, works namely how close to the actual value for the metric it is. This will also be tested with and without the presence of faults.

Lastly, to evaluate the accuracy of the fault detection implementation we will look into false negatives and positives detected, by introducing artificial faults into the replicas of the managed system.

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

In this document, we started by presenting the motivation and challenges that drives this work and also defining the goals for the work.

After that, we contextualized the problem showing the literature review done on the subject. First we presented some of the different state of the art protocols for byzantine fault tolerance and showed the differences between them. With these differences we concluded that neither of them is better than the others in every scenario. As such, it stands to reason that a system able to take advantage of the right protocol for the right context is an interesting concept.

With this in mind, we discussed Adaptive Systems, what they are, how they adapt and how they are composed. In this topic, we then discussed the monitoring target of our work, the replicated state machine, and looked into to what could be adapted in such a system.

Lastly we examined the target system of this work, the Monitoring System. Here we looked into how a monitoring system is composed, and how the monitoring target affects this architecture, namely how how the monitoring system needs to be fault tolerant in order to produce correct information for the Adaptation Manager. Furthermore, we discussed what the system needs to capture from the managed system and the topic of fault detection.

With the related work presented, we described our solution, taking into account the requirements that such a system would require and need to satisfy, namely introducing a byzantine fault tolerant architecture built with sensor extensibility in mind.

We close this report by presenting the metrics that will support the evaluation of our solution and assess the quality of our work.

Acknowledgments We are grateful to Carlos Carvalho, Miguel Pasadinhas, Daniel Porto, Manuel Bravo and Luis Rodrigues for fruitful discussions and comments during the preparation of this report. 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., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems **4**(3) (1982) 382–401
2. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating System Design and Implementation (1999) 1–14
3. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. Proceedings of the Twentieth ACM Symposium on Operating Systems Principles **39**(5) (2005) 59
4. 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, Berkeley, CA, USA, USENIX Association (2006) 177–190
5. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine Fault Tolerance. ACM Transactions on Computer Systems **27**(4) (2009) 1–39
6. Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. Symposium A Quarterly Journal In Modern Foreign Literatures (2009) 153–168
7. Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M., Aublin, P.l., Lyon, I.: The Next 700 BFT Protocols. Proceedings of the 5th European Conference on Computer Systems **32**(4) (2010) 363–376
8. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Prime: Byzantine replication under attack. IEEE Transactions on Dependable and Secure Computing **8**(4) (2011) 564–577
9. Bahsoun, J.P., Guerraoui, R., Shoker, A.: Making BFT Protocols Really Adaptive. In: Proceedings of IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015. (2015) 904–913
10. Bessani, A., Sousa, J., Alchieri, E.E.P.: State machine replication for the masses with BFT-SMART. In: Proceedings of 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014. (2014) 355–362
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7) (1978) 558–565

12. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. OSDI'04, Berkeley, CA, USA, USENIX Association (2004) 7–7
13. Padala, P., Hou, K.Y., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A.: Automated control of multiple virtualized resources. EuroSys'09 (2009) 13–26
14. Paiva, J., Ruivo, P., Romano, P., Rodrigues, L.: AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores. ACM Transactions on Autonomous and Adaptive Systems **9**(4) (2014) 1–30
15. 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)
16. Couceiro, M., Ruivo, P., Romano, P., Rodrigues, L.: Chasing the Optimum in Replicated In-Memory Transactional Platforms via Protocol Adaptation. IEEE Transactions on Parallel and Distributed Systems (2015)
17. IBM: IBM: An Architectural Blueprint for Autonomic Computing, 4th Edition. (2006)
18. Lamport, L.: The Part-Time Parliament. ACM Transactions on Computer Systems **16**(2) (1998) 133–169
19. Xu, Y., Musgrave, Z., Noble, B., Bailey, M.: Bobtail: Avoiding Long Tails in the Cloud. Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (2013) 329–341
20. Leverich, J., Kozyrakis, C.: Reconciling high server utilization and sub-millisecond quality-of-service. EuroSys'14 (2014) 1–14
21. Nightingale, E.B., Douceur, J.R., Orgovan, V.: Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. EuroSys '11 Proceedings of the sixth conference on Computer systems (2011) 343–356
22. Deepak Chandra, T., Thomas, t.J., Toueg, S., Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
23. Kihlstrom, K.P., Moser, L.E., Smith, P.M.M.: Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In Proceedings of the International Conference on Principles of Distributed Systems (1997) 61—-75
24. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Proceedings 10th Computer Security Foundations Workshop. (1997) 116–124
25. Doudou, A., Garbinato, B., Guerraoui, R., Schiper, A.: Muteness failure detectors: Specification and implementation. European Dependable Computing Conference (EDCC'99) **1667** (1999) 71–87
26. Denning, D.: An Intrusion-Detection Model. IEEE Transactions on Software Engineering **SE-13**(2) (1987) 222–232
27. Haeberlen, A., Kouznetsov, P., Druschel, P.: The case for Byzantine fault detection. Proceedings of the 2nd Conference on Hot Topics in System Dependability-Volume 2 (2006) 5–5
28. Leners, J.B., Hung, W.l., Aguilera, M.K., Walfish, M.: Detecting failures in distributed systems with the FALCON spy network. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11) (2011) 279–294
29. Leners, J.B., Walfish, M.: Taming uncertainty in distributed systems with help from the network. Eurosys (2015) 9:1—-9:16

30. Haeberlen, A., Kouznetsov, P., Druschel, P.: PeerReview: Practical Accountability for Distributed Systems. ACM SIGOPS Operating Systems Review **41**(6) (2007) 175
31. Duan, S., Levitt, K., Meling, H., Peisert, S., Zhang, H.: Byzantine Fault Tolerance from Intrusion Detection. Proceedings of the 33rd IEEE International Symposium on Reliable Distributed Systems (SRDS) (2014)