TÉCNICO LISBOA

Byzantine Fault Tolerant Monitoring of Distributed Systems

Bernardo Brito da Palma

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Doutor Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Doutor Luís Manuel Antunes Veiga
Supervisor: Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Doutor João Carlos Antunes Leitão

October 2017

Dedicated to my uncle.

Acknowledgments

I am grateful to Daniel Porto for the fruitful discussions and comments during the this whole process. I would also like to thank Carlos Carvalho, Miguel Pasadinhas and Manuel Bravo for the help given along this road. Lastly, I would also like to thank my family and friends for all the help and support given.

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.

Resumo

Os sistemas adaptáveis (SA) mudam o seu comportamento em resposta a variações no ambiente de execução causadas, por exemplo, por faltas ou alterações nos padrões de acesso. Um componente importante em qualquer SA é o sistema de monitorização (SM), responsável por recolher informações sobre a execução e detetar mudanças que justificam a adaptação. O SM é especialmente complexo na presença de faltas Bizantinas, onde elementos do sistema podem produzir mensagens incorretas, que por sua vez podem disparar adaptações indesejáveis, tornando o SA ineficiente ou vulnerável. Neste trabalho, descrevemos i) as escolhas realizadas na construção de um SM robusto e flexível para gerir vários tipos de sensores; ii) os mecanismos de consenso que garantem a obtenção de uma vista coerente do estado do sistema, agregando de forma tolerante a faltas as informações recolhidas pelos sensores. A avaliação demonstra que o nosso SM é facilmente extensível e tem capacidade de escala.

Palavras-chave: Tolerância a Falhas Bizantinas, Monitorização de Sistemas, Replicação de Máquina de Estados

Abstract

Adaptive systems (AS) are capable of altering their configuration in response to changes in its execution environment, caused, e.g, by faults or alterations in access patterns. A key component of any AS is the monitoring system (MS), responsible for collecting information regarding the operation and detect the changes to feed the policies that guide the system adaptation. The MS is especially complex in the presence of Byzantine faults, where system components may produce incorrect messages, or even trigger unwarranted adaptations, weakening the AS or making it less efficient. In this work, we describe i) the choices made in the development of a robust and flexible MS capable of handling various types of sensors; And, ii) the mechanisms that allow the system to provide a coherent view of the state of the AS, aggregating the information provided by the sensors in a fault-tolerant manner. The evaluation shows the extensibility of our SM and its scaling capability.

Keywords: Byzantine Fault Tolerance, System Monitoring, State Machine Replication

Contents

List of Tables

List of Figures

Listings

List of Abbreviations

AM Adaptation Manager. 14, 15, 23, 32, 38

API Application programming interface. 23, 26, 39, 41, 42

BFT Byzantine Fault Tolerant. 1, 5, 8, 9, 13, 15, 17, 18, 23, 24, 29

CFT Crash Fault Tolerant. 13

CPU Central Processing Unit. 14, 15, 17, 25, 47, 48

ES Event System. 16

IDS Intrusion Detection System. 20, 21

JVM Java Virtual Machine. 42

MonS Monitoring System. 1

MS Managed System. 24–27, 29, 31, 37

PBFT Practical Byzantine Fault Tolerance. 5–9, 13, 16

QCS Quality Control System. 16

RSM Replicated State Machine. 25, 26, 29, 44, 46, 49

SMR State Machine Replication. 11–13, 16, 17

Chapter 1

Introduction

1.1 Motivation

Adaptive systems modify their behavior in response to changes in their execution environment, such as faults, access patterns variations, utilization of shared resources, the variation in system workload imposed by the number of clients, service goals, etc. These dynamic changes trigger alterations that affect the performance of the different system's components. For example, some Byzantine Fault Tolerant (BFT) protocols like Zyzzyva [1] operate in favorable conditions, in a optimistic mode, where they present their best performance. That being said, in the presence of faults, Zyzzyva needs to execute complex additional stages of communication, that impose an higher number of messages, reducing its performance [2]. In similar fashion, other BFT protocols found in the literature also employ optimizations for specific scenarios of operation [3, 4, 5, 6, 7].

The lack of a standard solution, that is to say one adequate for every operation condition, created the opportunity for the construction of adaptive BFT systems that alternate between different protocols in response to the dynamic changes in the execution environment [6, 8, 9].

An important component of any adaptive system is its underlying Monitoring System (MonS), responsible for collecting information about the overall state of the main system, which can then be used to feed adaptation policies. This component is particularly complex in the presence of Byzantine faults, where incorrect replicas may produce messages in order to trigger wrong adaptations, making the system vulnerable to possible attacks or exhibiting sub-optimal performance. In order to avoid these issues, the MonS must also be tolerant to Byzantine faults. This means, not only being capable of tolerating incorrect MonS' replicas but also faulty sensors. Meaning that, even if a fraction f of replicas from a sensor produce wrong values, the MonS should still be able to feed the policies with correct and coherent information about the target system. Note also, that even correct replicas of a sensor might naturally diverge in the collected values. For example, the reading may be slightly shifted in time, meaning that although the values are correct, they are from different points in time. Furthermore, besides these challenges, its important that the MonS' architecture be flexible in order to develop more sensors.

1.2 Objectives

This work addresses the problem of designing and implementing a system capable of monitoring Byzantine fault tolerant systems. More precisely:

> This work aims at designing a redundant monitoring infrastructure that can provide consistent and accurate information regarding the operational environment, aggregated from different sources even in the presence of Byzantine faults.

1.3 Contributions

The dissertation makes two contributions:

- A software framework that simplifies the programming of fault-tolerant sensors.
- A set of fault-tolerant aggregation protocols that allow multiple replicas of the monitoring system to maintain a consistent view of the monitored system despite the presence of faulty sensors.

1.4 Results

The following results have been achieved:

- An implementation of a replicated, Byzantine fault-tolerant, monitoring system, using the open source BFT-SMaRT system.
- A comparative evaluation of the different fault-tolerant aggregation protocols in local and geo-replicated settings.

1.5 Research History

This work is part of a research project named Abyss with the purpose of developing a Byzantine tolerant system capable of adapting to the conditions of its execution environment. As such, it is part of a larger collaboration currently divided into three fronts, the adaptable/managed system, the adaptation manager and the monitoring system (the target of this thesis). Furthermore, this builds on a previous contribution by Frederico Sabino[10]. We benefited from the fruitful collaboration of Daniel Porto, who was always available for discussion and giving ideas throughout this thesis. A paper that describes an early version of our work proposed in this dissertation has been published in Actas do 9º Simpósio de Informática, Inforum 2017 [11].

1.6 Thesis Outline

This thesis is structured as follows. In Chapter 2 we present the context upon which this work builds upon. In Chapter 3 we start by presenting the assumptions that guide the development of our system. After we present the architecture and go into detail about the different components that make up the system. The results of the evaluation performed are presented in Chapter 4. Finally, in Chapter 5 we conclude this thesis and present possible future improvements.

Chapter 2

Background

In this chapter we will present and discuss the relevant related work found in the literature that supports and motivates this thesis. First, we start by presenting some relevant BFT protocols and highlight their differences (Section 2.1). Then, we discuss what and how are adaptive systems composed (Section 2.2), and what can be adapted in replicated state machines (Section 2.3). We continue by looking into BFT system monitoring (Section 2.4) and present some examples of their aplications in adaptive systems (Section 2.5). Lastly, we conclude the chapter by presenting what is relevant to monitor with said component (Section 2.6) and then more concretely discuss fault detection (Section 2.7).

2.1 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.

2.1.1 PBFT

The protocol known as Practical Byzantine Fault Tolerance (PBFT)[3] is the first BFT protocol 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[12]. 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:

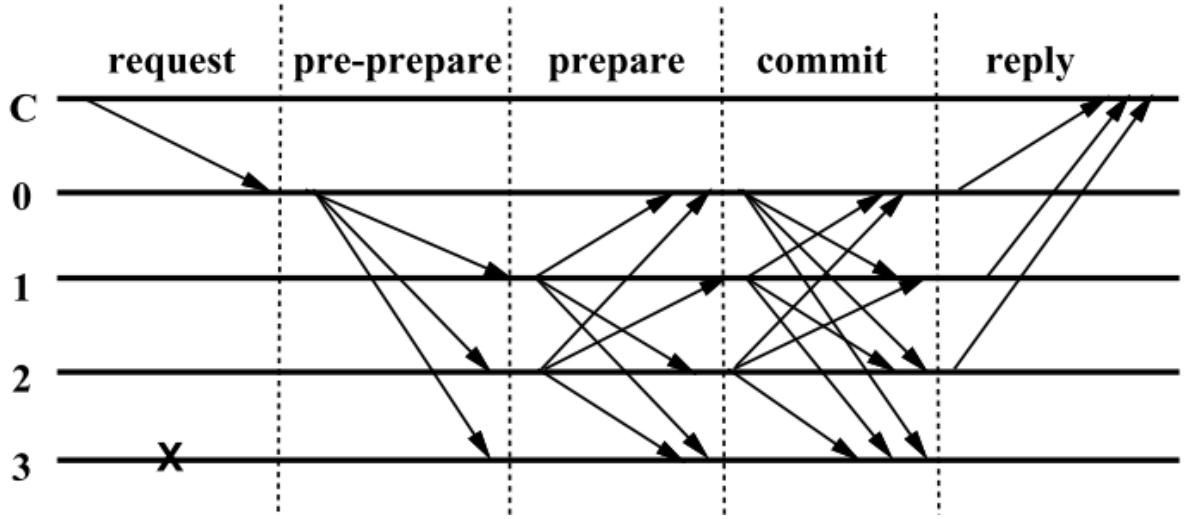


Figure 2.1: PBFT's Normal Case Operation; f=1, replica 3 is faulty. Taken from [3].

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 2.1 shows the operation of the algorithm in the normal case scenario (not that this is agnostic to the presence of faults).

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

2.1.2 Zyzzyva

Zyzzyva[1] 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 2.2 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 Misbehaviour* 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 process 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

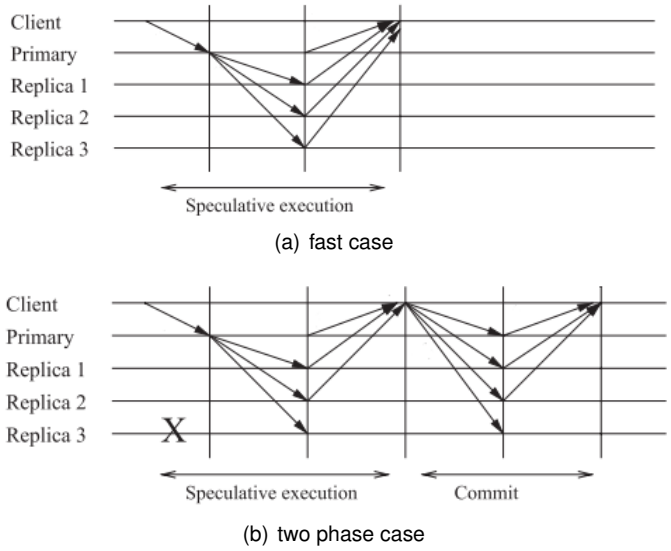


Figure 2.2: Zyzzyva Message Patterns; f=1. Taken from [1].

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.

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.

2.1.3 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 [6] 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

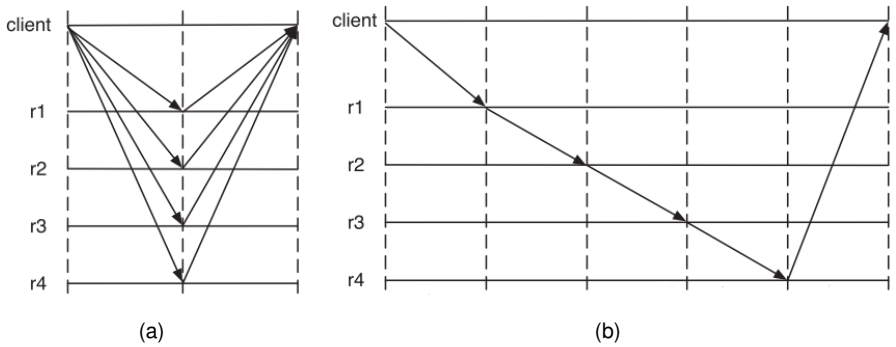


Figure 2.3: Quorum (a) and Chain (b) Message Pattern; f=1. Taken from [6].

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 2.3(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[13] 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 2.3(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.

2.1.4 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 occurrences. 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.

2.2 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[14, 15] (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[9], 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, 8].

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, Analise, 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 2.4. 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 adapta-

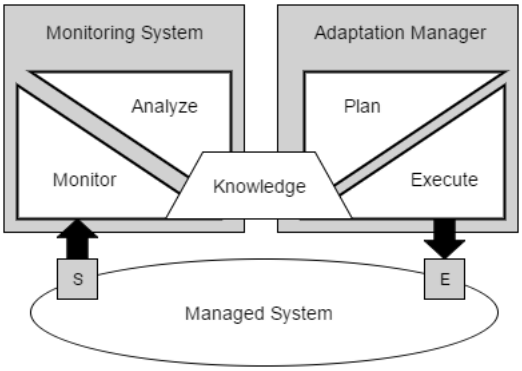


Figure 2.4: Generic Adaptive System Architecture.

tions, 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.

2.3 Adaptation of Replicated State Machines

In this section we 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.

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 2.1 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 that 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.

Analysing this type of system and looking at an actual implementation, namely *BFT-SMaRT*[18] 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;

2.3.1 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 changes in the system's environment.

While the system is processing 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 [9] 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.

2.3.2 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 Tolerant (CFT)* or *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[19], 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.

2.3.3 Protocol Reconfiguration

As discussed in Section 2.1, there are a wide range of different BFT 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 [6, 8]. 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[6]. 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 [8]. 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.

2.3.4 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[9] or even by moving replicas to other machines in a more "secure" location.

Also, considering that the system can only make guarantees 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[20, 21] or even if the probability of failure for that machine has increased[22].

2.4 Byzantine-Tolerant 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 can detect.

2.4.1 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 Central Processing Unit (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.

2.4.2 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, it 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 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 thesis we advocate a deployment where each replica of the monitoring broker is collocated with each replica of the adaptation manager, such that the fault-tolerance of these two components can be dealt in an integrated manner. However, the need to replicate 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 these replicas.

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 sensor 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 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 (excluding malicious attacks), or that, for some metrics, the adaptation policy is robust to inaccurate readings, and therefore less than $3f + 1$ replicas can 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.

2.5 Monitoring in Adaptive BFT Systems

Like it was discussed before, some adaptive BFT systems have already been introduced, each with its own monitoring component. As such, in this section we will discuss the different solutions that have

been introduced in said systems.

2.5.1 Aliph

Alyph[6] is a pioneer of its kind, an adaptive state machine system capable of switching its underlying Byzantine fault tolerant protocol as its context changes. With this system, the current protocol function only in the conditions it was designed to perform best and when those conditions change it defers the execution to another protocol that can handle them, where after a quarantine period the initial protocol is put in place.

The system changes protocols in a fixed “circular” order utilizing three different protocols to handle worsening degrees of concurrency or asynchrony deferring in the end to PBFT[3] as a means to handle the worst case scenarios. This static protocol switching presents some disadvantages as it does not take into account other information from the environment that also affects the protocols’ performance (e.g. latency or number of clients), and as such it presents a barely existent monitoring infrastructure, where only when a timeout is reached or inconsistency with the replicas responses occurs, the current protocol initiates a PANIC process and the next one in line is put in effect. Returning back when a quarantine period has passed.

2.5.2 ADAPT

Adapt[8] builds upon this concept gaining in performance against the previous system by collecting and utilizing more information about the context of the managed system in order to chose the next best protocol, thus making the system capable of dynamically switching its underlying protocol.

Adapt is divided into two main components, namely a Quality Control System (QCS) and an Event System (ES). The QCS takes care of utilizing the information collected to evaluate the current protocol and deciding if switching to another would grant the managed system more performance or if that gain in performance is worth the switch. In order to evaluate the protocols, it uses machine learning models to predict the protocol’s performance. The ES is the component in charge of collecting information from the system’s context that most impacts the system and feeding it to the QCS.

Since the main focus of this work falls on the QCS and the dynamic protocol switching the ES is left as possible future work. Thus, it is implemented as a simple module, only reading information from the application, such as message size and the number of active clients, and not employing any form of fault tolerance, not being robust to the presence of faulty sensors or its own replicas.

2.5.3 Bytam

Bytam[9] tries to expand the adaptability of these types of systems by introducing more flexibility in the development of adaptation policies, namely introducing the possibility to define *Event-Condition-Action* policies. Furthermore, besides flexibility it also expands the targets of said adaptations, not only considering changes in protocol but also changes to the number of replicas or internal parameters of the SMR.

Although in this system, and contrasting with ADAPT, a robust monitoring infrastructure is considered where a replicated component that can receive information from replicated sensors, the main focus of the paper is still the adaptation side of the system showing a limited specification for the monitoring system and a rigid monitoring infrastructure restricted to a fixed set of sensors.

2.6 Relevant Metrics to Monitor

The relevance of different metrics 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. 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. These 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[8]. 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 of 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).

2.7 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 (if $f = 1$). 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 2.1, 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.

2.7.1 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. [23] 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.[24] and Malkhi et al.[25], to name a few.

Unreliable Failure Detectors have been categorized as having two properties[23]:

- *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

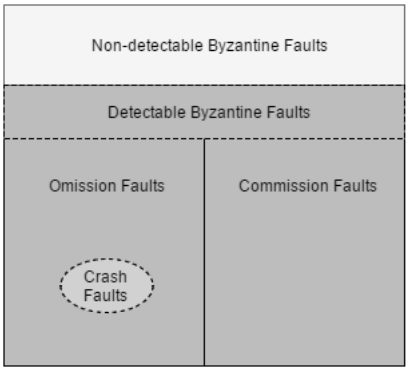


Figure 2.5: Byzantine fault classification.

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.[26] 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[24].

- *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 2.5 we present a visual representation of this fault classification.

Considering that the presence of these faults can also happen 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*[27]. 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.

2.7.2 Interest in Fault Detection

Haerberlen et al.[28] 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[29] 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[30] 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[31] 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[32], 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 do this by integrating an instance of said IDS into each replica to monitor and discard messages that are not in conformity 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.

Summary

In this chapter we looked into to relevant work in the literature from where we derived important requirements and knowledge that will be used through the specification and implementation of a robust monitoring system.

The next chapter will introduce the architecture and implementation details of our system.

Chapter 3

Architecture and Implementation

This chapter describes the architecture of our BFT monitoring system, the provided sensor development/deployment Application programming interface (API) and implementations of the consensus module experimented.

We start by presenting the assumed system model followed by a general overview of the system's architecture. Then we decompose the system and explain the functioning and responsibilities of each of the components present in the system. In the consensus module the implementations that will be part of the focus of the evaluation are presented.

3.1 System Model

We assume a distributed system composed by several processes that communicate by exchanging messages. The system is asynchronous, in the sense, that processing and communication times may occasionally exceed any limitation previously imposed. However, we also assume that the behavior of a majority of the processes, including the ones pertaining to sensors, does not make a timely adaptation to the system an infeasible task. The processes also have access to an external source of time. This source is used to mark the different readings with timestamps; we also assume that the clock synchronization of the processes pertaining to the sensors is external to our system and done by a trusted source.

The different components of the system are subject to the occurrence of Byzantine faults, thus capable of exhibiting arbitrary behavior. These faults may occur due to natural causes or from the intrusion of malicious agents. We assume that a maximum $f = \lfloor \frac{n+1}{3} \rfloor$ replicas of each component may exhibit Byzantine behavior, where n is the total number of replicas for the given component; these replicas may also collude in order to try subverting the system. Lastly, we also assume that the processes possess limited resources and can not break the cryptographic techniques used in our algorithms.

Lastly, although the system could feed different types of systems, we currently assume that the target that consumes the monitoring information is an AM integrated with our system that uses the collected data to evaluate policies and trigger adaptations in order to improve a managed system. We also assume that this manager is deterministic in its actions and decisions.

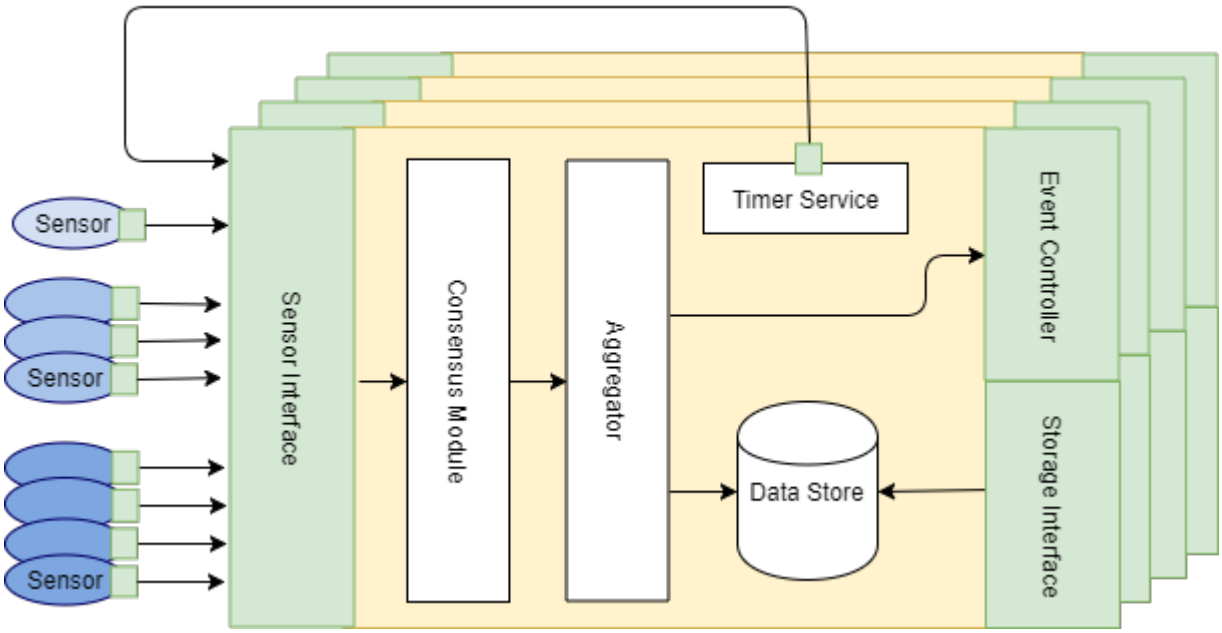


Figure 3.1: System Architecture

3.2 System Overview

The monitor system is comprised of four main components, namely: A set of 1.*Sensors*, a Monitoring Broker which we designate by 2.*Aggregator*, a 3.*Data Store* and a 4.*Consensus Module*. And a few secondary ones, that provide useful functionality not tied to the core functioning of the system. A general representation of the architecture is presented in Fig. 3.1.

The sensors collect information from the Managed System (MS) and its execution environment. They work in one of two operation modes: they may respond to a sporadic event (e.g. a crash, leader change) or provide a continuous flow of updates regarding some characteristic from the MS, to compute statistics (such as system load or throughput). Given that a BFT system is replicated, for robustness, the sensors also ought to be replicated, whose collected data can then be compared and/or aggregated. Each sensor can monitor one or more replicas. As a consequence of multiple sensors, replicated data can be generated for the same event or reading. This replicated data may be shifted, this can happen due to benign reasons such as different sensors detecting the same event in slightly skewed moments in time, or because of a faulty sensor reporting wrong sensing data. Thus, an *Aggregator* service is responsible for gathering the data from all sensors and solve inconsistencies in their perspective of the event, after it is passed through a *Consensus Module*. Finally, the processed sensor data is stored in robust and consistent manner in a *Data Store* to enable the design of policies from data aggregated across several sensor probes and longer periods of time. We detail these components in the next sections.

Table 3.1: Sensor's Properties

Identifier	Uniquely identifies the single sensors ou group of sensors (replicas)
Replica Identifier	Identifies the different replicas of a sensor (when applicable, by default zero)
Cardinality	Size of the group of sensor replicas
Number of Faults	Fault limit under which the sensor (or group) can operate correctly
Operation Mode	Reactive or Periodic
Rate	Rate at which it collects new values (applicable only to Periodic sensors)
Fault Model	Non replicated, Crash Tolerant, Byzantine
Credentials	Cryptographic (asymmetric) keys to authenticate the sent messages

3.3 Sensors

The *Sensors* have the function of collecting information from the MS and its execution environment. The various sensors can be configured to have different fault models, namely they can tolerate crash faults, tolerate Byzantine faults or not tolerate any faults. The type of fault model chosen will depend on a number of factors, like the semantics of a certain data to be collected or the possibility of installing different independent sensors to observe a given phenomenon. For example, the throughput of the MS or number of active clients can be seen across its different replicas and, as such can be collected by independent sensors, allowing for Byzantine fault tolerance for these metrics. On the other hand, some sensors possess characteristics that does not allow this or where its replication does not make sense (e.g CPU statistics from a server).

Each sensor possesses a set of properties necessary in order to identify and process the information collected. Said properties are presented in Table 3.1.

The data collected from the MS is passively sent to the aggregator in order to be processed. Since, and as it will be discussed ahead, the aggregator is implemented as an application on top of a Replicated State Machine (RSM) the sensors are similar to clients sending commands to the RSM. As such, these can be sensors themselves or merely proxies for the actual probes, allowing them to also "encapsulate" possible legacy sensors. However, contrary to regular RSM clients, the sensors do not need to wait for a response from the aggregator as they do not perform actions upon the system.

Each message sent to the aggregator by the sensors contains the collected information associated with a locally attributed sequence number, incremented at each new message. Furthermore, the message also contains the sensor identifier, the identifier of the respective replica and a timestamp, identifying when the collection was made. This way the replicas can be grouped as single sensor in the aggregator. Lastly, all sent messages are signed with the respective credentials of the sensor.

We classify the collected type into two different categories: *Metrics* and *Events*. We define metrics as numeric values periodically collected that represent a specific information about the MS (e.g some host's CPU load). On the other hand, we classify events as "arbitrary" situations that occur in the MS, such as a leader change or the detection of a fault in one of its replicas.

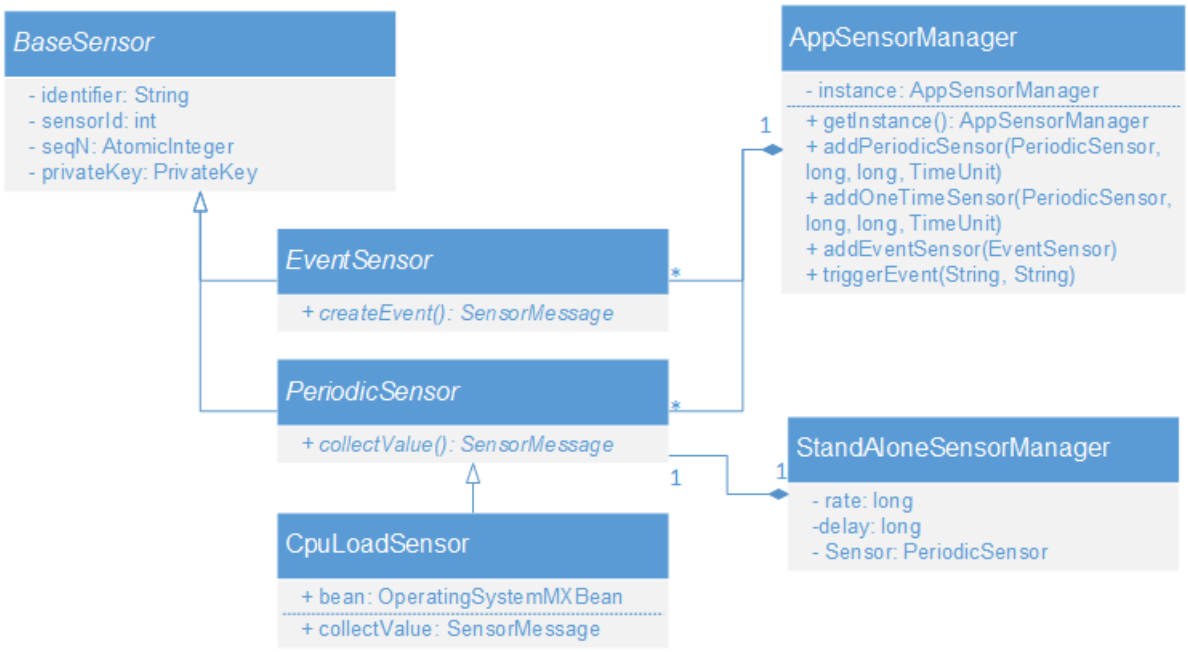


Figure 3.2: API Class Diagram

3.3.1 Extensibility

The managed system's goals may be subject to change, introducing new needs and consequently new possible adaptations. This may imply the need to gather new metrics from the MS' execution environment, meaning the sensor infrastructure would need to expand in order to accommodate these new requirements. As such, a small API is presented in order to facilitate the development of new sensors and also abstract the communication details between the sensor client and the RSM. Furthermore, this API also offers some deployment mechanisms for the sensors by default, as this is also one of the main efforts of extending the sensor infrastructure or initializing them at system startup. The general structure of thisAPI is presented in Figure 3.2.

3.3.2 Sensor Deployment

For sensor deployment we assume two possibilities. Either the sensor needs access to internal information pertaining to the MS or it needs to collect information about the MS' execution environment, meaning external to its core execution. As such, the sensor needs to be deployed as a concurrent execution thread along with the MS or it can be deployed as a separate process, respectively. In order to help in the deployment of the different sensors we introduce two "managers", namely *AppSensorManager* and *StandAloneSensorManager* both of them work with sensors that implement the sensor interfaces that will be presented below.

The first one, is a component that is embedded into the MS allowing sensors to be added/registered. Periodic sensors can be added along side their rate and an optional delay, and the manager will take care of launching the sensor and performing the collections with the provided rate. These sensors can be recurrent meaning they do more than one collection or they can be deployed to fire a single time, for

Listing 3.1: Abstract Class PeriodicSensor.

```
1  package argus.sensors;
2
3  ...
4
5  public abstract class PeriodicSensor extends BaseSensor{
6
7          public PeriodicSensor(String identifier, Integer sensorId, PrivateKey pKey)
               {...}
8
9          public abstract SignedMessage collectValue();
10
11 }
```

example gathering the MS' replica id. For reactive sensors the manager works a bit differently. Since events may be triggered for a variety of reasons, its difficult to create an expressive enough interface. As such, when a reactive sensor is registered the manager, instead of launching a sensor, merely stores the sensor information, creating something similar to a registry entry. Then a method is provided to be called upon the occurrence of the event, that receives the value and sends it using the registry information for that particular sensor. Lastly, in order to make the same and only manager instance assessable through the MS, it is implemented as a singleton.

The second one, although it is called StandAloneSensorManager (for consistency), it is in fact a simple process launcher. Currently, it only supports the deployment of periodic sensors. When executed, it receives the name of the sensor to be deployed, its replica identifier, the rate at which it will collect the metric and a possible delay until its first collection, and starts as a process with the purpose of collecting the metric from the chosen sensor.

3.3.3 Sensor Development

For periodic sensors we offer an abstract class, *PeriodicSensor* shown in Listing 3.1, that can be extended and requires the implementation of the collection method and have its constructor called to initialize the information pertaining to the sensor, namely identifier, replica id and private key. An example of a possible sensor implementation is presented in Listing 3.2. This interface along with the deployment options presented above allow the abstraction of some communication details/concerns for the developer.

As discussed before, for reactive sensors we currently only provide a way to create something to hold the sensor/replica information and create a correct signed message to be sent (using the stored information) by the AppSensorManager when required. This is shown in Listing 3.3.

3.4 Aggregator

In order to simplify the sensors implementation and for the monitoring system to be robust, the *Aggregator* was developed as a replicated service, extending a fairly popular open-source state machine

Listing 3.2: Example of PeriodicSensor implementation.

```
1  package argus.demo.sensors;
2
3  import ...;
4
5
6  public class CpuLoadSensor extends PeriodicSensor {
7
8          OperatingSystemMXBean bean;
9
10         public CpuLoadSensor(int sensorId){
11                 super("CpuLoad" + sensorId, sensorId, SecurityUtils.getPrivateKey(
                       Constants.SENSORS_HOME + "/" + "CpuLoad" + sensorId + "/keys/
                       private0.der"));
12                 this.bean = (OperatingSystemMXBean) ManagementFactory.
                       getOperatingSystemMXBean();
13         }
14
15         @Override
16         public SignedMessage collectValue() {
17
18                 BigDecimal result = new BigDecimal(bean.getSystemCpuLoad() * 100);
19
20                 System.out.println("SystemCPULoad: " + result);
21
22                 MetricMessage value = new MetricMessage(getSequenceNumber(), 0,
                       getType(), System.currentTimeMillis(), result);
23
24                 return new SignedMessage(value, getPrivateKey());
25         }
26 }
```

Listing 3.3: EventSensor Class.

```
1  package argus.sensors;
2
3  import ...;
4
5  public abstract class EventSensor extends BaseSensor{
6
7          public EventSensor(Integer sensorId, String identifier, PrivateKey
               privateKey){
8                  super(identifier, sensorId, privateKey);
9          }
10
11         public SignedMessage createEvent(String value){
12                 EventMessage em = new EventMessage(getSequenceNumber(), 0,
                       getIdentifier(), System.currentTimeMillis(), value);
13                 return new SignedMessage(em, getPrivateKey());
14         }
15 }
```

replication library called Bft-SMaRt[18], that is already capable of tolerating Byzantine faults. Since the sensors act as clients to this RSM, the collected values received through their messages can be totally ordered by the underlying BFT protocol, thus creating a linear sequence of values and allowing all correct replicas of the aggregator to work on them by the same order. This allows the sensors (or more specifically their replicas) to only concern themselves with the collection of the data, leaving the responsibility of processing and consolidating that information from the MS to the aggregator.

As values are being received and before they can be processed and later stored or delivered to, for example, an adaptation manager to feed its policies, the aggregator must first await minimum quantity of messages, namely a quorum, for each sequence number. The size of the quorum for a particular sensor is extrapolated from the previously referred properties. Note that, quorums higher than 1 are only strictly required for sensors set to tolerate Byzantine faults. Although some benefits could be had from aggregating messages of several replicas of Crash tolerant sensors (mitigating slight divergences in values), a single message can be used as it will be a correct representation of the collection.

3.4.1 Aggregation Function

After the quorum of values from a particular sensor has been reached, it needs to be condensated into a singular value, representative of the state of that particular data point, furthermore, this is required to allow the aggregator to decide on a timestamp for the collection done. To do this, each sensor or set of sensor' replicas has an aggregation function associated with it, that is applied to the group of values accumulated. These functions need to be deterministic and can be basic operations like averaging the obtained values or be ones capable of filtering Byzantine values, like the one offered by the system for example, removing the f lowest and f highest readings and averaging the resulting values (f being the number of faults by that particular sensor), Listing 3.4. An interface is provided in order to allow new functions to be introduced, e.g adaptations of existing ones present in the literature [33, 34]. Although not strictly necessary to aggregate non-replicated sensors' collected values, the function can also serve to do some preprocessing on the gathered value.

Since the BFT protocol ensures that each correct replica the aggregator processes the same collected values by the same order and the aggregation function is deterministic, all correct replicas will reach the same aggregated result, thus achieving robust aggregation.

Lastly, it should be noted that the values obtained by applying these functions are what we designate by approximately correct values since, due to small divergences, correct sensor replicas may collect slightly different values. For example, if the replicas of a given sensor collect the values 40, 43 and 45, we assume that any value in the range of 40 to 45 is approximately correct and a representative of that particular collection.

3.4.2 Sensor Information Registration & Manipulation

The aggregator itself is agnostic in regards to the implementation of the sensors. In order to correctly function, it only requires some knowledge regarding the properties of the sensors deployed along with

Listing 3.4: Example of a Fault tolerant aggregation function.

```
1  package argus.aggregator.function;
2
3  import ...;
4
5  public class FaultTolerantFunction implements AggregationFunction<BigDecimal>{
6
7          @Override
8          public AggregatedValue<BigDecimal> execute(SensorMessage<BigDecimal>[] input
               , int f, SensorType sensorType) {
9
10                 BigDecimal[] in = new BigDecimal[input.length];
11                 long[] ts = new long[input.length];
12
13                 for(int x = 0; x < input.length; x++){
14                         in[x] = input[x].getContent();
15                         ts[x] = input[x].getTimestamp();
16                 }
17
18                 Arrays.sort(in);
19                 Arrays.sort(ts);
20                 BigDecimal[] values = Arrays.copyOfRange(in, f, in.length - f);
21                 long[] timestamps = Arrays.copyOfRange(ts, f, in.length - f);
22
23                 BigDecimal result = new BigDecimal(0);
24
25                 for(BigDecimal v : values){
26                         result = result.add(v);
27                 }
28
29                 return new AggregatedValue<>(input[0].getIdentifier(), sensorType,
                       result.divide(new BigDecimal(values.length), RoundingMode.FLOOR)
                       , timestamps[0]) ;
30         }
31
32         @Override
33         public Boolean validate(Object input) {
34                 return input instanceof BigDecimal;
35         }
36 }
```

Listing 3.5: Example of an aggregation config file

```
1 identifier=Throughput
2 sensorType=Metric
3 f=1
4 quorum=3
5 aggregationFunction=argus.aggregator.function.FaultTolerantBigDecimalAverageFunction
```

how they should be aggregated and the reception values using the correct format. As such, the system provides two mechanisms to register this information, namely:

1. The properties from the sensors may be defined in a file called *aggregation.config* (an example is provided in Listing 3.5) along with a folder containing the different public keys pertaining to the sensor. Having these been defined, the system will automatically load them into the aggregator during its initialization.

2. A sensor may be registered after the system has been initialized though a provided interface. This allow some flexibility to the developer, allowing this data to be retrieved from an external source or even be hardcoded into the system.

As stated in Section 3.3.1, the sensor infrastructure may need to be expanded or reduced and furthermore, some adaptations, e.g replica relocation, as they are triggered, may shutdown or redeploy certain sensors or their replicas as a result of it, possibly changing configuration parameters of said sensors, like quorum size or aggregation function. As such, the second mechanism can also be used for this purpose. In order to alter the parameters for a particular sensor or remove it, the method pertaining to that information needs to be called while providing the correct unique identifier and the new value to be introduced. Currently, this is only accessible by each of the adaptation manager's replicas collocated with each of the replicas of the aggregator.

3.5 Data Store

Although adaptation policies may be triggered due to recent or immediate events, some may require a view of the MS over a large period of time. This would allow those policies to understand, e.g workload tendencies or heavier access periods, thus allowing the creation of more complex adaptations and preemptively triggering them in order to optimize the system. Furthermore, it would allow the mitigation of the effect of transient out of norm collections and the possible combination of different types of metrics and events.

In order to allow these type of policies the system, after aggregating the sensors' collections, stores the final values in a *Data Store* and provides access to them through a *Storage Interface*, discussed ahead. Each of the replicas of the aggregator will have a copy/instance of the data store, effectively making this also a replicated component, as you would expect. Since each replica of the aggregator reaches the same values by the same order, it will produce a consistent and coherent state across each data store.

Each entry in the data store contains the value collected/aggregated, its type, the corresponding timestamp (for when the collection was made) and an entry id, whose purpose will be explained ahead.

The prototype of this component is implemented as a relational database embedded in the system using H2[35]. For a more concrete/complete implementation the use of a proper time-series database such as InfluxDB[36] or KairosDB[37] would be beneficial as it would allow a broader and more complex manipulation of data natively.

3.5.1 Storage Interface

This interface's purpose is, as you would expect, allowing the target system to access the stored information in order to feed its policies. Since the data store is currently implemented in H2, SQL is the language utilized for accessing the data.

In order for the data to be accessed in a robust and coherent manner by all correct replicas of the adaptation manager, the AM needs a way to limit it's search. As such each entry recorded in the data store is marked with a unique id (as of now, merely a increasing counter) that serves as a version number for the store. Each time the AM is "called into action", either due to an event or a timer being triggered, the id of the latest version is passed as an argument which can then be used to limit the search. This guarantees that all correct replicas of the AM will evaluate its policies using the same input data, since it has a single entry point activated by ordered events.

For a different type of target system, namely one that runs concurrently with our own, this measure would not suffice, such that additional mechanisms would need to be added. For example, all accesses to the data store would need to be coordinated and passed through the consensus and most likely the end system would need to also employ some coordination in its implementation (if replicated).

3.6 Consensus Module

As mentioned before, the aggregator groups the values collected by the sensors in order to guarantee that every correct replica executes operations over the same coherent view of the data. The sensors' messages are associated with one another in accordance with the properties specified for the sensor or replicated group, registered in the aggregator. In a similar fashion, a lower boundary on the amount of messages needed for each type of sensor is defined based on the quantity of tolerated faults, cardinality and type of fault model selected, as shown in section 3.3. The messages can be ordered as they arrive or accumulated first for later ordering. Thus we can divide this process into two steps, *Accumulation* and *Consensus*. We experiment with four different approaches in order to implement this module and report on their performance in section 4.2.

3.6.1 Post-Consensus Accumulation

In the first implementation, designated as PosC, each value received from the sensors is immediately totally ordered through the consensus, namely the Bft-SMaRT library. Only after being delivered to the

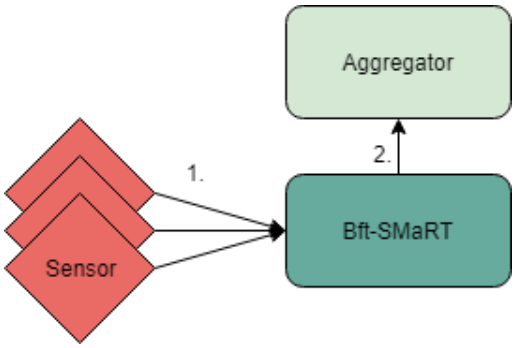


Figure 3.3: Message pattern for PosC and Integrated-PreC.

aggregator for accumulation and eventual aggregation upon reaching the quorum defined for the sensor. As such, in this case, we have the consensus step before the accumulation one. This means that even for replicated sensors, a consensus is performed for each value/message sent by the sensor or sensor's replicas (this ignoring possible batching). Upon delivering the ordered value to the aggregator, its validity is tested with the following steps:

1. Verify that the sensor with the same identifier as the value received is registered in the aggregator;
2. Validating the authenticity of the message by checking its correct signature by the sensor or its replica;
3. Check value freshness;
4. Finally check if the quorum has been reached.

This is the base implementation and is the one considered while explaining the different components and their responsibilities described in the rest of the sections in this chapter. A visual representation of the general message pattern for PosC is presented in Figure 3.3 (the internal message exchange of Bft-SMaRT is omitted[38]) and the internal process in Figure 3.4.

3.6.2 Pre-Consensus Accumulation

In this section we will discuss the remaining implementations, Total-PreC, Disperse-PreC and Integrated-PreC, which execute the Accumulation step before ordering.

Implementations: Total-PreC & Disperse-PreC

For the first two, the sensors send their values directly to the aggregator to be accumulated, without a defined order. During this accumulation, each of the values passes through the verification described in the previous section. When the quorum is achieved, the accumulation is sent by an internal client to be totally ordered, thus the consensus is finally run. This implementation choice adds an extra communication step to the protocol when compared to the one presented in the previous section, as shown in Figure 3.5. After the set of values is ordered, it is delivered back to the aggregator to be aggregated,

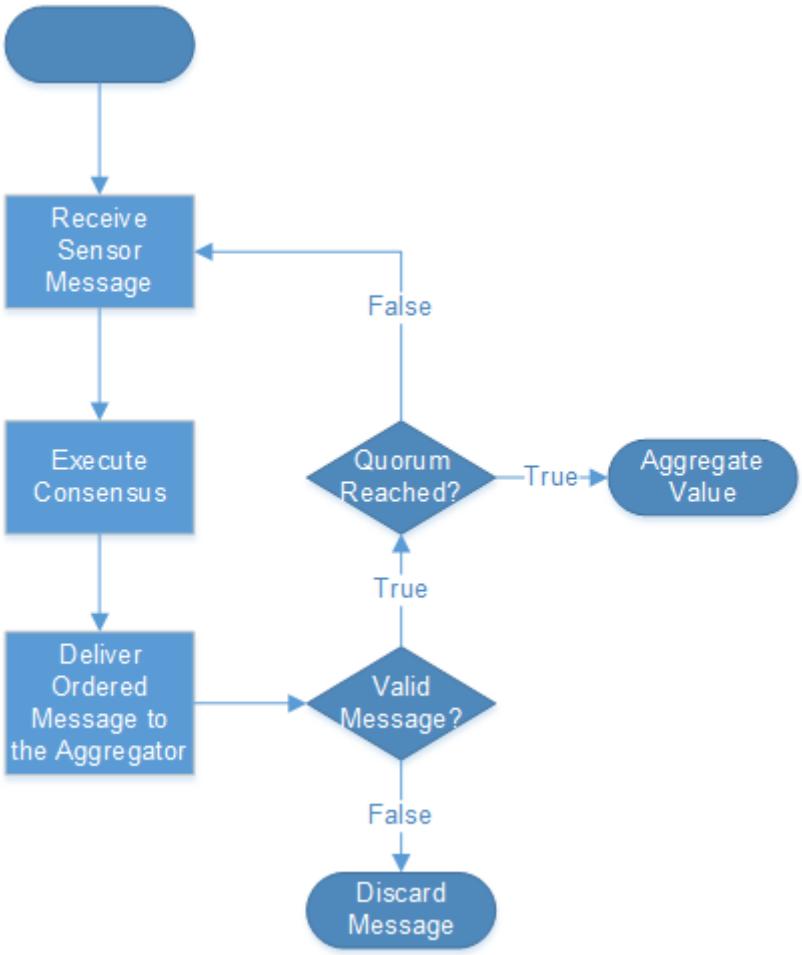


Figure 3.4: Flow diagram for the PosC implementation.

passing first through a similar verification step as before to guarantee its correctness after ordering, namely:

1. A sensor with the same identifier is registered in the aggregator;
2. The set quorum for that sensor has indeed been reached;
3. The messages/values accumulated are correctly signed;
4. They are from “unique” sources, meaning the values are not accumulated from the same replica;
5. The message freshness, namely if the sequence number has not been previously decided;

In variant Total-PreC, every correct replica sends the accumulated value set as soon as the quorum is achieved. As such, the accumulations are effectively replicated, not being necessary to have a leader replica (excluding obviously the underlying consensus protocol). When one of these messages passes through the final verification, after being ordered, the remaining ones are discarded. This means that the number of consensus executed does not differs from PosC, thus presenting a clear cost. Even in the best case scenario, this implementation wastes a great deal of work done, due to the replication of the messages sent to the consensus, being latter discarded. On the other hand, it shows a simple

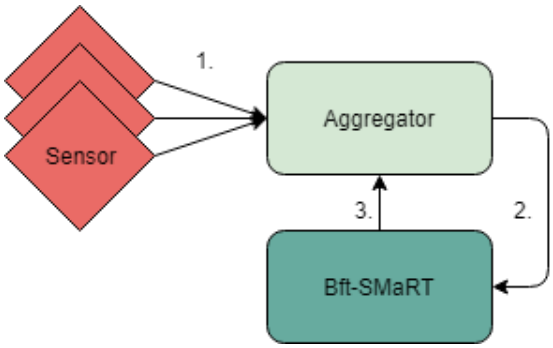


Figure 3.5: Message pattern for Total-PreC and Disperse-PreC.

to implement pre-consensus accumulation, without altering the replicated state machine and explicitly dealing with the possible occurrence of faults.

The second variant, Disperse-PreC, tries to improve the previous one namely, it tries to avoid running a consensus for each value received. In order to do that, the role of sending the accumulations to the consensus is distributed between the replicas, by applying a hash function to the unique identifier of the different sensors. As such, each of the sensors is attributed to a certain replica. In the optimal case, without the occurrence of Byzantine faults, each of the replicas of the system is only in charge of sending the accumulations for the sensors its responsible for. With this optimization the load is then distributed amongst the replicas. However, there is the possibility that the progress of certain sensors will be affected when one of the replicas becomes Byzantine, either by sending wrong values or by not sending them at all. To deal with this cases, when the correct replicas detect a lack of progress for a specific sensor, they revert to the first implementation, Total-PreC, for the sensors attributed to the faulty replica. The flow diagram for this implementation is presented in Figure 3.6, this also shows the internal workings for Total-PreC if the sensor assignment verification was removed.

Integrated Implementation: Integrated-PreC

The objective of this implementation is to remove the extra communication step that arises from the two previous solutions. In order to do this, the aggregator was merged with the Bft-SMaRT library, and as such, the accumulation step was integrated into the underlying consensus protocol.

The messages sent by the sensors are received by the library similar to what happens in implementation PosC, but the message is intercepted before the consensus is initiated and passed through the first validation process previously described, thus the value is retained. And as such, that message itself is never ordered. When the quorum is reached, a message is created internally with the accumulated values and the consensus is initiated. As such, the intermediary step of using an internal client to relay the message is cut from the process and the extra communication step is thus removed. Since the messages from the sensors can be received in different orders by the replicas of the system, the accumulation that is used by the leader to start the consensus may be different to the one reached by the remaining replicas. As such, in order to not trigger unnecessary leader changes, messages are not compared based on content, but on the identifier of the sensor and the respective sequence number.

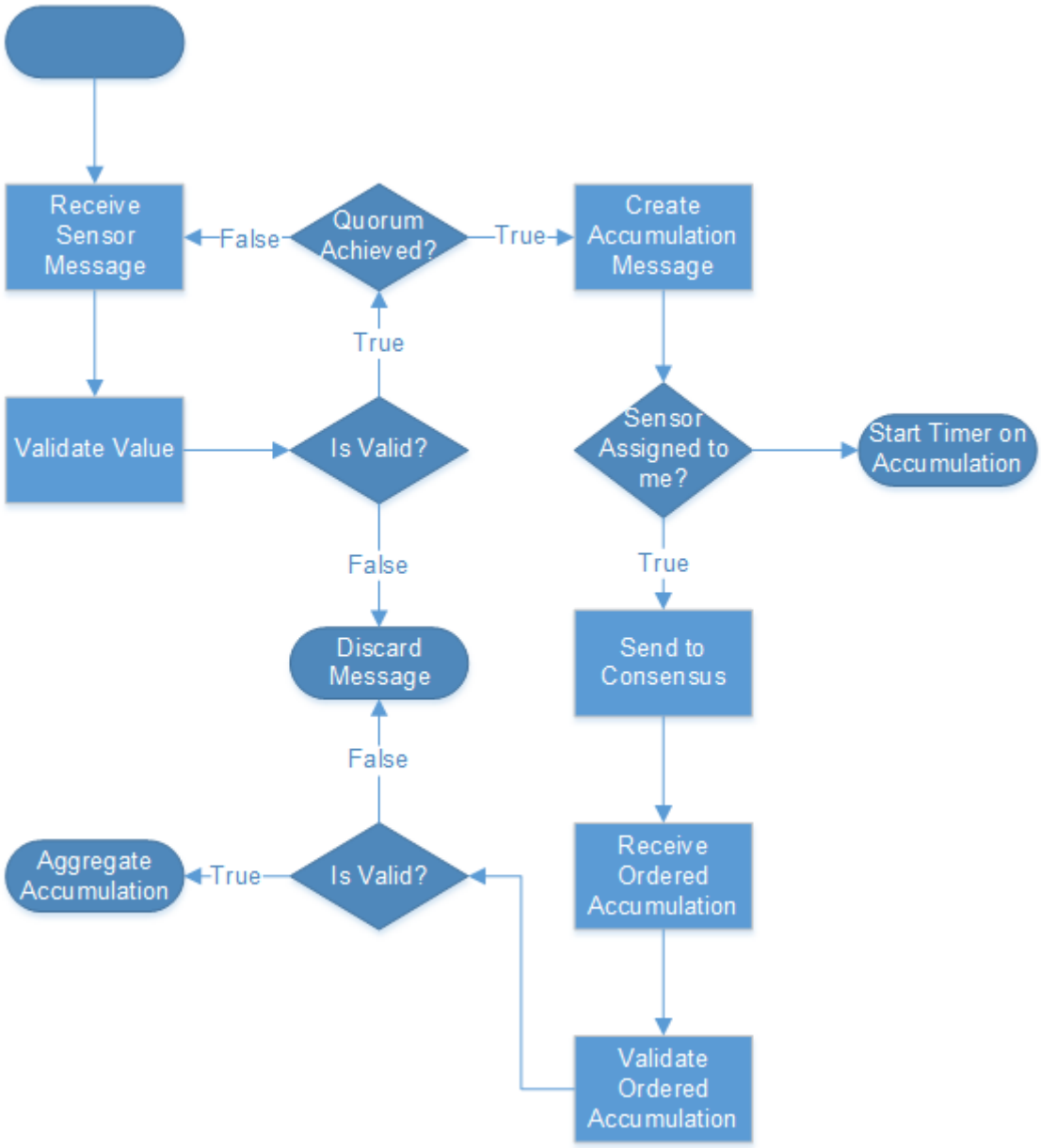


Figure 3.6: Flow Diagram for Disperse-PreC.

Instead of the content being compared, to guarantee correctness and keep the current leader replica in check, the second validation step refereed previously is done during the communication steps of the underlying protocol. Namely, its run by each replica upon receiving the propose that starts the consensus. If this step fails, the leader change procedure is started. After being totally ordered the accumulations are then aggregated.

Lastly, an optimization is employed for sensors that only require quorums of one message. Running the two full validations is unnecessary in this scenario, when a message from these sensors is received, only a check for singular quorum is done, before and during the consensus. Only after being ordered and delivered is the value validated using the first validation refereed, minus the last step of checking a reached quorum. As such, this implementation shows a similar message exchange as PosC, shown in Figure 3.3, but does away with the need to run a consensus for every value received, which is relevant

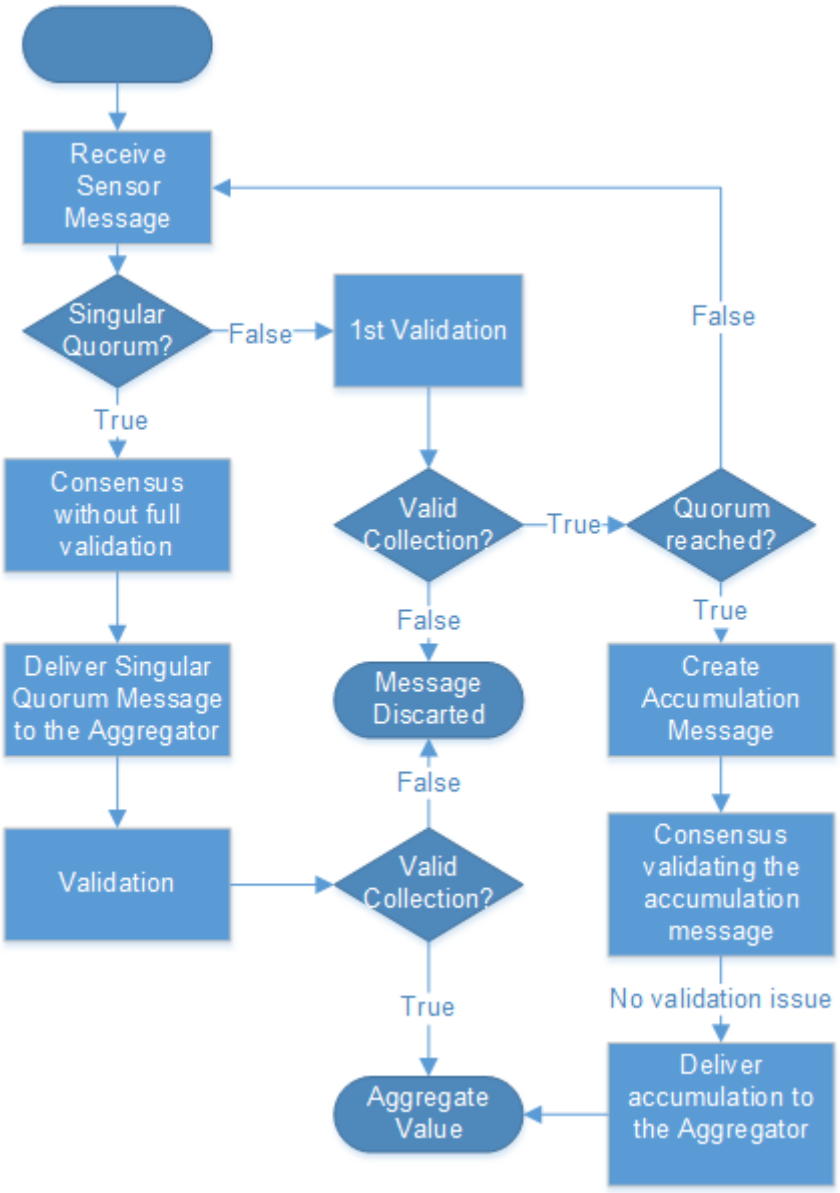


Figure 3.7: Flow diagram for the Integrated-PreC implementation.

for replicated sensors. The flow diagram is presented in Figure 3.7.

3.7 Other Components

Besides the main components discussed above, the system has other components, that while not being crucial to its core functioning, provide necessary features. Those will be discussed in this section.

3.7.1 Timer Service

An adaptation manager, besides consuming information pertaining to the MS, namely metrics and events, may require to also track specifics periods of time during its operation, for example to verify the effectiveness of a previously triggered adaptation or when to evaluate the general state of the target

system and deciding if some policy should be activated. As such, the monitoring system also provides a *Timer Service* that allows multiple timers to be set. This is a replicated service where each replica is collocated with the each replica of the aggregator, thus having the same replication factor. As such, the notifications from this service only occur when the timers are triggered in a majority of replicas and their events have passed through the consensus.

The timers work as clients to the replicated state machine, using an internal client. These send events to the system as they are triggered and, as it happens with the sensors, these values pass through the aggregator where an accumulation occurs. Finally, when a quorum is gathered, the timer event may be delivered. In this case, the condensation of the several values is done by a no-op aggregation function, seeing as the important information is the triggering of the timer.

The service provides two types of timers, (1) One time timers, that are triggered after a certain degree of delay specified at its creation, useful to reach an agreement on whether a grace period of a certain adaptation has finished. And (2) Periodic timers, that, as the name implies, trigger from time to time, and whose rate is defined at creation. Besides the parameters, delay and rate, mention above, each timer also has associated an unique identifier attributed at creation, allowing specifics actions to be triggered based on it.

Furthermore, the service makes use of the registration interface of the aggregator in order to add and remove timers from the registered sensor list. Lastly, due to the assumptions made about the adaptation manager, this addition and removal of timers is done individually at each of the replicas of the service by the respective replica of the AM. Similar to what was discussed in section 3.5.1, if we were to consider a wider range of target system types, the management of timers would need to be done through the use of consensus and also with the target system possessing some internal coordination.

3.7.2 Event Controller

Although events detected by sensors are also saved in the data store, they are a type of information that require in the general case an immediate reaction. Thus when they occur, some specific action may need to triggered in the AM related to the that event, e.g if a leader change is detected the AM may be required to chose the next one. As such, the *Event Controller* allows the registration of *handlers* associated to specific events/sensors, namely their identifier. When an event is captured and properly aggregated, the respective handler is called, if it has been previously registered.

Lastly, this component may also be used to register handlers for the timers currently active in the system, starting, for example, the policy revaluation by the AM or retrieval of values from the data store.

Summary

The specification for the architecture of our system was presented in this chapter, along with its implementation. The different components of the system were described regarding their functioning and responsibilities. Finally, we also presented other implementations for the consensus module with the

focus on improving its performance.

In the next chapter we present the experimental evaluation made by analyzing the sensor development/deployment API and the different implementations for the consensus module.

Chapter 4

Evaluation

With the evaluation in this thesis we seek to answer two main questions:

1. How helpful is the provided interfaces for the extension of the sensor infrastructure?
2. How do the implementations for the consensus module stack against one another in terms of performance?

As such, this Section is divided into two parts, a Qualitative Evaluation that presents the answer to the first question and second a Quantitative Evaluation that tests the performances of the different implementations.

4.1 Qualitative Evaluation

As stated above, this section will discuss the usefulness of the provided sensor development/deployment interfaces. That being said, due to some constraints no proper user based evaluation was performed. As such, this discussion will be more of a “theoretical” one performed by us, albeit as objective and impartial as possible.

As evidenced by the class diagram presented in Figure 3.2, the API does not have the complexity of something like JDBC, as it presents only a few interfaces. The ones offered allow the separation of the collection concern from the deployment of the different sensors. For example, if a developer wanted to implement a similar sensor to the one presented in Listing 3.2 to be deployed as a separate process from the MS without using the API it would look similar to the one present in Listing 4.1. Although not really complex, there is a clear difference in the development effort for this sensor using the API and without. Furthermore, if it was developed to be deployed as a concurrent thread in the MS, besides either extending the Thread class or implementing the Runnable interface, alterations to the MS would need to occur in order to actually deploy said sensor, either by just starting the thread or using a scheduler. This difference in effort becomes even more apparent if we consider that in the most common case scenario the developer would probably need to develop more than a single sensor, or even further expand the

sensors as requirements change. As such, the API provides some features that would most likely end up being implemented by the developer himself.

The most weak aspect of this API is without a doubt the reactive sensor support. In terms of integrated deployment it does present some base functionality through the triggerEvent method, where an event/alert can be sent as its triggered in the internal code of the application. That being said, it does lack the support for the deployment of this type of sensor as a separate process. In order to remedy this, besides improving the StandAloneSensorManager, some interfaces could eventually be developed, for example a threshold sensor, that periodically checks some information and send an event when it reaches a certain limit, or a connected sensor, that sleeps until it receives a message from some source and relays it as an event.

Concluding, although the API is simple and could clearly be further developed, it does provide a good base for sensor development with some useful features.

4.2 Quantitative Evaluation

4.2.1 Evaluation Setup

The system and sensors used in the evaluation were hosted by Digital Ocean[39]. Each replica of the system had its own individual virtualized environment, while the sensors shared some hosts as it will be explained ahead. The specifications for the virtualized machines of both the system's replicas and sensors is presented in Table 4.1. The notable difference in specification is not with the intention of running a sensor more powerful than the system, but several sensor threads in a single environment mitigating the occurrence of bottlenecks.

Table 4.1: Virtualized Environments' Specifications

	System replica's	Client/Sensors'
CPU (cores)	4	8
RAM (GBs)	8	16
SSD (GBs)	80	160

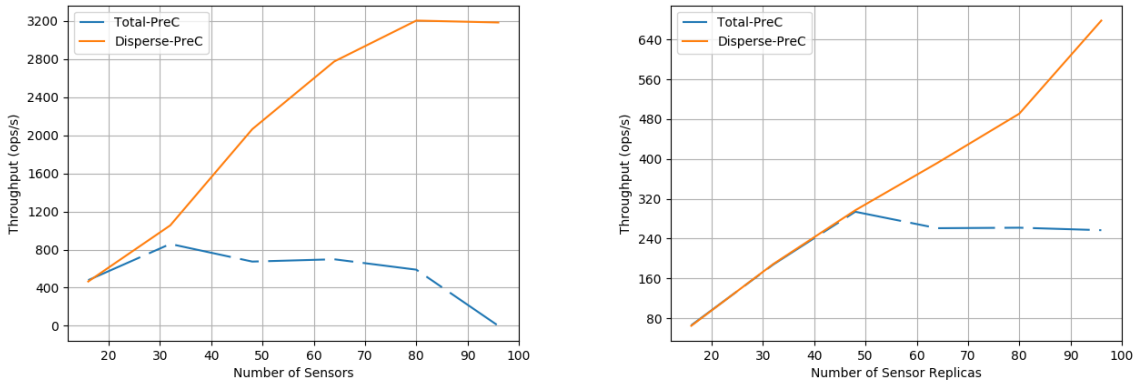
Considering that our system was implemented resorting to the Bft-SMaRT library, the components are executed in a Java Virtual Machine (JVM). The version used to run the system/sensors was 1.8.0_144. Each replica was launched without changing the default JVM's heap size values.

4.2.2 Evaluation Performed

In order to evaluate the performance of the different implementations, they were subjected to varying degrees of workloads. In the experiments, a value is considered decided after it is aggregated, as such, in order for the complexity of the aggregation function to not affect the results, a dummy one was utilized merely returning the first value from the obtained set. Furthermore, the aggregated values are not stored in the data store to merely test the performance of the algorithms. Considering that the objective is to analyze the limits of each pre-consensus accumulation versions and compare them with the base

Listing 4.1: Cpu Load Sensor implemented without the API

```
1  package argus.demo.sensors;
2
3  import ...;
4
5  public class CpuSensor {
6
7          public static void main(String args[]) throws IOException {
8
9                  if (args.length < 2) {
10                         ...
11                 }
12
13                 Integer seqN = 0;
14                 Integer processId = Integer.parseInt(args[0]);
15                 Integer typeId = Integer.parseInt(args[1]);
16                 String type = "CpuLoad" + typeId;
17                 PrivateKey privateKey = SecurityUtils.getPrivateKey("sensors/" +
                       type + "/keys/private0.der");
18
19                 ServiceProxy sProxy = new ServiceProxy(processId, "monitor-config");
20
21                 OperatingSystemMXBean bean = (OperatingSystemMXBean)
                       ManagementFactory.getOperatingSystemMXBean();
22
23                 Double warmup = bean.getSystemCpuLoad();
24
25                 while(warmup < 0){
26                         warmup = bean.getSystemCpuLoad();
27                 }
28
29                 BigDecimal result;
30
31                 while (true) {
32
33                         try {
34                                 Thread.sleep(1000);
35                         } catch (Exception e) {
36                                 e.printStackTrace();
37                         }
38
39                         result = new BigDecimal(bean.getSystemCpuLoad() * 100);
40
41                         System.out.println("SystemCPULoad: " + result);
42
43                         MetricMessage value = new MetricMessage(seqN, 0, type,
                               System.currentTimeMillis(), result);
44                         SignedMessage message = new SignedMessage(value, privateKey)
                               ;
45
46                         sProxy.invokeOrdered(SerializableUtil.serialize(message));
47
48                         seqN++;
49
50                 }
51         }
52 }
```



(a) (b)

Figure 4.1: Performance Comparison of Total-PreC and Disperse-PreC. With non-replicated sensors 4.1(a) and replicated ones 4.1(b).

implementation, in which the accumulation is done post consensus, the tests will be synthetic in the workload that they will generate. Thus, these do not represent real world scenarios. In the deployment of the monitoring system, we defined $f = 1$, thus generating four system replicas.

The sensor used is based on the micro-benchmark already offered by the RSM library. Each sensor/replica is deployed as an independent thread that consistently sends the same correctly signed value, without any wait occurring between the messages sent. This is done in order to achieve a high workload. In order to create a consistent load during a considerable amount of time, this value is sent a total of 8000 times by each sensor. In this evaluation, we vary the amount of sensors deployed and alternate between replicated and non-replicated. This is done in three different scenarios, namely, a no latency (or a negligible amount) scenario between the different environments, a flat latency scenario and a emulation of a real world latency scenario. The last two will be performed only with replicated sensors as that is the main case we want to evaluate, namely if saving on running a consensus per value yields any performance gain. Furthermore, these latencies were introduced using Unix's *tc* command.

The implementations that will be tested will be PosC, Disperse-PreC and Integrated-PreC. Total-PreC was put aside for the testing as the optimized version, Disperse-PreC performs better, as evidenced by some preliminary testing done shown in Figure 4.1. Note though, that for these the client machines had the same specifications as the system machines. Its also important to refer that the hash function used in implementation Disperse-PreC distributes the sensors uniformly between the different replicas.

For the experiments the final throughput values were obtained by performing the median between the average of values reached by the each of the 4 replicas, before any sensor had finished sending messages. This allows the results to not be affected by transient higher/lower values.

For the tests with non-replicated sensors, 16 sensors are launched concurrently in each client machine, while for the replicated ones only 4 sensors are deployed. Note though, that each one of these last ones is in fact composed of 4 independent threads, thus each client machine in either test generates an equivalent amount of message load for the system. In both tests, five experiments were conducted, increasing the number of client hosts from 1 until 5, making up 16, 32, 48, 64, 80 and 4, 8, 12, 16, 20

sensors for the non-replicated and replicated tests, respectively.

Note as well that from the results a complete direct comparison cannot be made for the different scenarios, as these were performed at different point in time and although the machines have the same specifications, their performance vary as the conditions of the physical host for the virtual environment changes.

Lastly, for the different experiments the timeout value for the timer that tracks the liveness of the system (more concretely the one that makes sure the leader makes progress) is set to an “infinite” value in order for it to not affect the results. This is justifiable, as in our particular testing we are interested in the best case scenario where no faults occur. Furthermore, since every replica is located inside the same data center we can expect that no message is lost and as such the leader is capable of receiving all of them.

Flat Latency Experiment

With this experiment, we wanted to emulate an equidistant network with a considerable amount of latency. As such, we introduce 50ms of latency with a jitter of 3ms between the system’s replicas and co-locate the sensors with the leader. The topology is similar to the one presented in Figure 4.2, but the values are all 50ms except the link between sensor and leader which is the same.

Real World Latency Experiment

The scenario we chose to emulate was a Wide Area Network spread across 4 different places of the globe, as such the latencies introduced are based on the average latencies between Amazon EC2 regions taken from [40]. For this test we chose the regions Frankfurt (where the leader is placed), Tokyo, Sydney and North California. Once again the sensors are co-located with the leader replica and a topology for this experiment is presented in Figure 4.2.

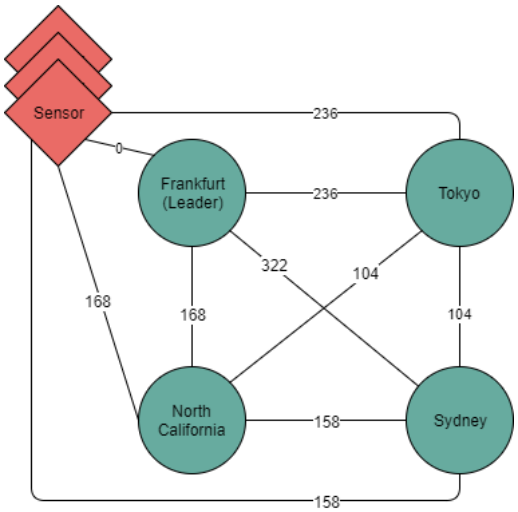


Figure 4.2: Real World Latency Scenario Topology. Values of latency presented as ms.

4.2.3 Non-Replicated Sensor Results

Although the main focus of the implementation Disperse-PreC and Integrated-PreC is to try improving the system's performance with replicated sensors, we conducted an experiment with non-replicated sensors as stated before. The results are presented in Figure 4.3. As we expected, the difference in performance between the base implementation PosC and the integrated version Integrated-PreC is not significant, as the implementations are similar in the way that they handle non-replicated sensors' messages.

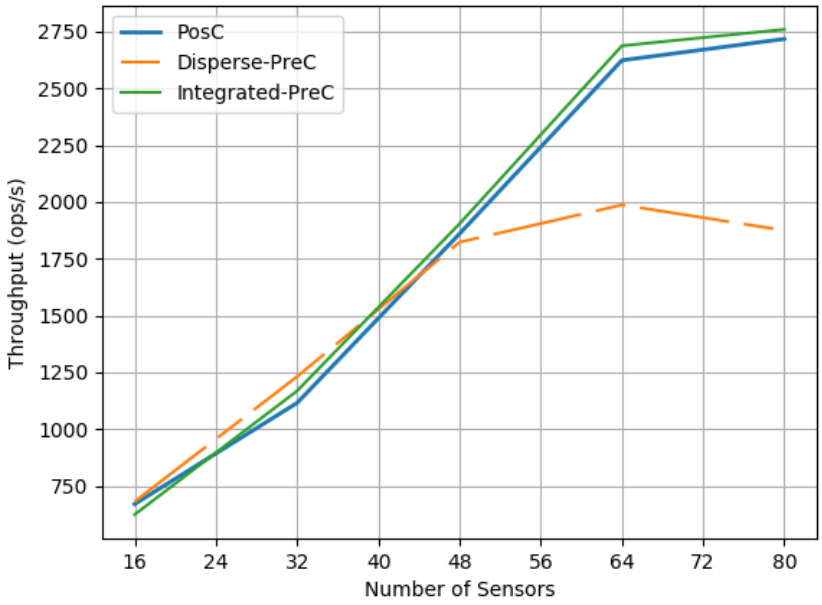


Figure 4.3: Performance results with non-replicated sensors in a latency free scenario.

Implementation Disperse-PreC after the third test is incapable of keeping up with the other two implementations showing a clear difference in throughput in the following tests. Furthermore, in the final test with 80 sensors one of the replicas ended up not being able to conclude the test, due to an exception regarding the exhaustion of memory in Java's heap space. That being said this was the least performant replica as shown across the different tests. This is due to the fact that this implementation ends up working as a re-sender for each message sent by the sensors due to the extra communication step and as such its overwhelmed by the load introduced in the final tests. An optimization could be introduced to fix this issue, either by having the sensors being aware of their non-replicated characteristic and sending their values directly to the RSM for ordering instead of sending it to the aggregator for accumulation first, or by changing the underlying RSM to handle these cases similar to how Integrated-PreC does it. The first option would break the separation between sensor and monitoring system, and the second one would go against the point of this implementation of not altering the RSM, and as such non of the two is ideal for the purpose of this evaluation.

4.2.4 Replicated Sensor Results

These results are the ones we are most interested in discussing as they are the main focus of these implementations. In a latency free environment, results shown in Figure 4.4, all three implementations present similar performance for most of the tests. As expected, implementation Disperse-PreC is now capable of handling the higher quantities of incoming messages since it only needs to send the accumulation to the consensus. In the last test with 20 sensors, small differences in throughput appeared, namely with Integrated-PreC. This difference foreshadows something that becomes apparent in the next scenario with flat latency introduced.

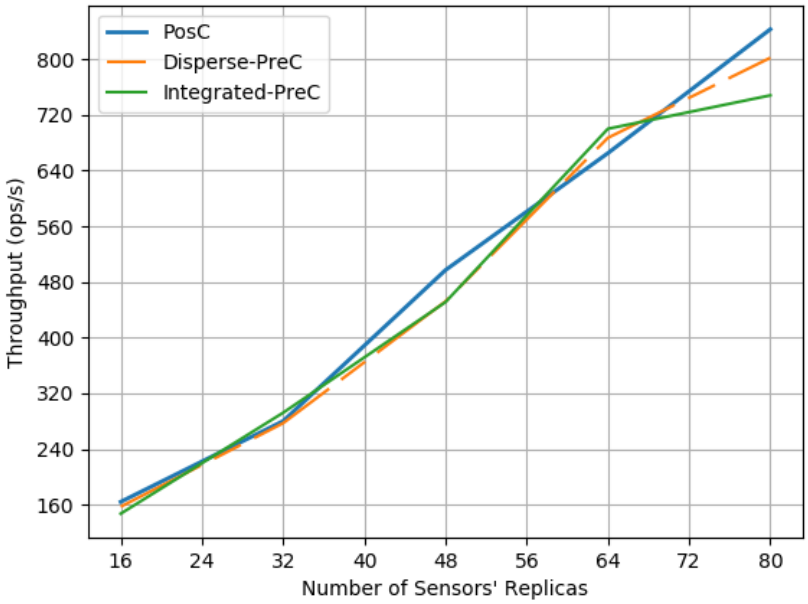


Figure 4.4: Performance results with replicated sensors in a latency free scenario.

As shown in Figure 4.5, implementations PosC and Disperse-PreC present no real differences in performance in any of the tests performed, on the other hand implementation Integrated-PreC starts showing lower performance with 12 sensors onwards until the last test where it seems to hit a cap at roughly 720 ops/sec. This was not what we expected, with this scenario we were hoping to see this implementation show a level of performance either on par or slightly better than the remaining implementations. This seems to occur because in this scenario the consensus is still more bound by CPU than by latency, and since Integrated-PreC runs a more expensive validation during the consensus after the reception of the propose, it thus achieves a lower throughout. This finding is supported by Disperse-PreC showing similar performance to PosC although its presents an extra step in communication and also performs the same secondary validation step but not during the consensus process.

With the real world scenario, we expected that the consensus would be more bound by latency, as it was increased, allowing for the implementation Integrated-PreC to gain performance comparatively to PosC. In part this is true, as shown in Figure 4.6, Integrated-PreC does for a moment present higher performance than PosC, albeit not really a significant difference. In the last test with 80 sensors both

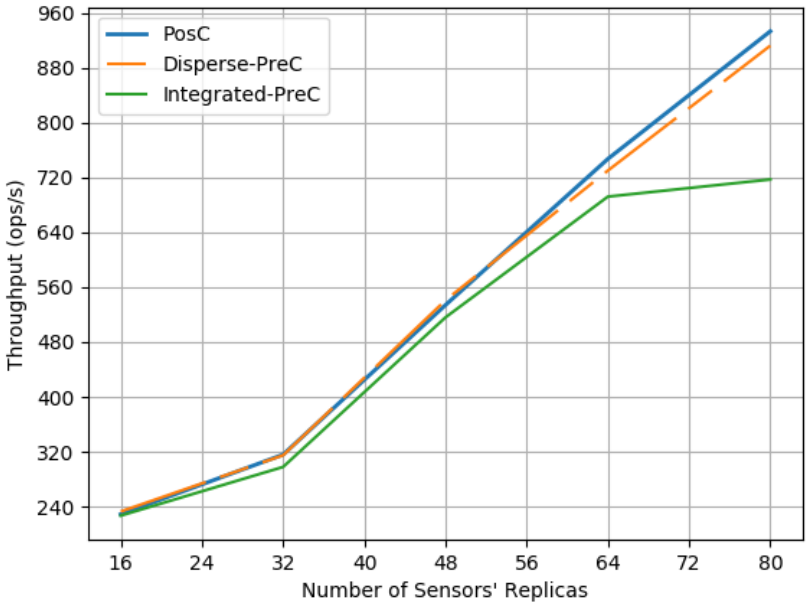


Figure 4.5: Performance results with replicated sensors in a flat latency scenario.

these implementations showed a drop in throughput. The big surprise in this scenario is Disperse-PreC that is able to handle the workload from the last test and present higher performance than the remaining implementations. This occurs because its consensus is not as bound by CPU as the one from Integrated-PreC and due to the accumulation that allow it to run fewer consensus than PosC, even with an extra communication step. This shows that accumulating the values before running the consensus has advantages against running a consensus per value. Furthermore, considering that in the previous scenario Integrated-PreC was showing less throughput than PosC, with the increase in latency it was able to achieve slightly higher performance than PosC showing the benefits of pre consensus accumulation.

4.2.5 General Discussion

Although the results were not exactly what we expected, where the integrated solution would present the best results, the general idea of applying the accumulation step before running the consensus did show positive results which was part of our hypothesis. From these experiments, we could also see some room for possible improvements in these implementations. For example, for Integrated-PreC we could try to improve the validation step that is done after the propose phase of the consensus and resolve any inefficiency present or, instead of using Asymmetric Encryption to sign the messages, the sensors/replicas could use it to exchange Symmetric keys helping to improve performance. Furthermore, an hybrid implementation between Disperse-PreC and Integrated-PreC could be attempted, resolving the issues of Disperse-PreC with non-replicated sensors and possibly improving performance with replicated ones.

Lastly, for real world utilization any of these implementations would perform similarly, as the tests conducted do not represent this scenario, since even the workloads from the tests with the least amount

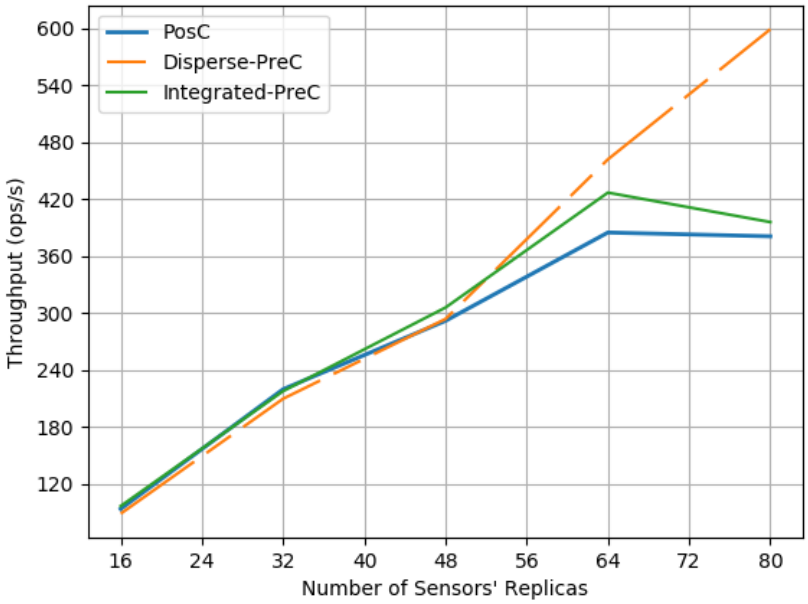


Figure 4.6: Performance results with replicated sensors in a real world latency scenario.

of sensors would still surpass most cases of normal use. That being said, as of now in this context, the best implementation would probably be PosC, since the system is built upon a proven to work stock RSM library where most, if not all, corner cases have been considered, thus possibly providing the most stable deployment.

Summary

In this chapter the evaluation of our work was presented along with its results. We started by looking into the provided sensor development/deployment API and then into the performance of the implementations for the consensus module that were developed.

The next chapter closes this thesis by presenting the conclusions regarding the work developed and also introduces some possible routes in terms of future work.

Chapter 5

Conclusions

5.1 Conclusion

This thesis was motivated by the base idea of creating a Byzantine fault tolerant system that can adapt itself according to its execution environment and its changes. An integral component of these types of systems is their monitoring infrastructure, meaning how do they perceive their context. This is then the focus of our work.

As such, we presented the architecture of a monitoring system not only capable of tolerating Byzantine faults of its core components, but also capable of tolerating such faults of its sensors, allowing them to present varying degrees of replication. More concretely, we presented the different components of the system, their functionalities and how they interact with each other. Furthermore, an API was presented in order to help develop and deploy the required sensors and abstract the implementation details of their communication with the system.

Some experimentations were also made with regards to how the consensus and accumulation work. In the base implementation, the consensus occurs first, and the accumulation is only done after the values are ordered, for replicated sensors it means that each value sent by the replicas passes through a consensus. As such, some implementation were presented with the accumulation step occurring first and only then passing the full group of values through the consensus.

Lastly, we presented the evaluation performed, firstly looking into the strengths and shortcoming of the provided API and then the performance differences of the different implementations for the consensus and accumulation steps.

5.2 Future Work

In term of possible future steps, we would like to improve upon a few aspects:

- Further develop the provided API to accommodate more sensors and more customization;
- Currently the sensors and the remaining components of the system, namely the aggregator, are

treated almost as separate identities, for possible future work it would be interesting to try and add the concept of life cycle to the sensors and give the ability for the central system to control that cycle, as this would allow for a more seamless/automatic reconfiguration procedure. This does bring some other issues in terms of coordination between the different changes in configuration, where introducing the concept of views could be helpful;

- Lastly, as discussed in section 4.2.5 some improvements could be attempted to the presented implementations. Furthermore, more experiments could be performed, for example repeat the ones performed but spread the sensors' replicas across the network.

Bibliography

[1] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, Jan. 2010. ISSN 0734-2071. doi: 10.1145/1658357.1658358. URL http://doi.acm.org/10.1145/1658357.1658358.

[2] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558988.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://dl.acm.org/citation.cfm?id=296806.296824.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095817. URL http://doi.acm.org/10.1145/1095810.1095817.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298473.

[6] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 363–376, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755950. URL http://doi.acm.org/10.1145/1755913.1755950.

[7] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Secur. Comput.*, 8(4):564–577, July 2011. ISSN 1545-5971. doi: 10.1109/TDSC.2010.70. URL http://dx.doi.org/10.1109/TDSC.2010.70.

[8] J. P. Bahsoun, R. Guerraoui, and A. Shoker. Making bft protocols really adaptive. In *2015 IEEE*

International Parallel and Distributed Processing Symposium, pages 904–913, May 2015. doi: 10.1109/IPDPS.2015.21.

[9] F. Sabino, D. Porto, and L. Rodrigues. Bytam: um gestor de adaptação tolerante a falhas bizantinas. In *Actas do oitavo Simpósio de Informática (Inforum)*, Lisboa, Portugal, Sept. 2016.

[10] F. Sabino. Bytam: a byzantine fault tolerant adaptation manager. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Sept. 2016.

[11] B. Palma, D. Porto, and L. Rodrigues. Monitorização de sistemas tolerantes a faltas bizantinas para suportar adaptação dinâmica. In *Actas do 9º Simpósio de Informática (Inforum)*, Aveiro, Portugal, 2017.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21 (7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL `http://doi.acm.org/10.1145/359545.359563`.

[13] R. van Renesse and F. B. Schneider. 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, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251254.1251261`.

[14] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519068. URL `http://doi.acm.org/10.1145/1519065.1519068`.

[15] J. a. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 9(4):19:1–19:30, Dec. 2014. ISSN 1556-4665. doi: 10.1145/2641573. URL `http://doi.acm.org/10.1145/2641573`.

[16] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-6471-3. doi: 10.1109/DSN.2013.6575311. URL `http://dx.doi.org/10.1109/DSN.2013.6575311`.

[17] A. Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31, 2006.

[18] A. Bessani, J. a. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-2233-8. doi: 10.1109/DSN.2014.43. URL `http://dx.doi.org/10.1109/DSN.2014.43`.

[19] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL http://doi.acm.org/10.1145/279227.279229.

[20] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482658.

[21] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592821. URL http://doi.acm.org/10.1145/2592798.2592821.

[22] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer pcs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966477. URL http://doi.acm.org/10.1145/1966445.1966477.

[23] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43 (2):225–267, Mar. 1996. ISSN 0004-5411. doi: 10.1145/226643.226647. URL http://doi.acm.org/10.1145/226643.226647.

[24] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *In Proceedings of the International Conference on Principles of Distributed Systems (OPODIS*, pages 61–75, 1997.

[25] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 116–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7990-5. URL http://dl.acm.org/citation.cfm?id=794197.795085.

[26] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, EDCC-3, pages 71–87, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66483-1. URL http://dl.acm.org/citation.cfm?id=645332.649834.

[27] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, Feb. 1987. ISSN 0098-5589. doi: 10.1109/TSE.1987.232894. URL http://dx.doi.org/10.1109/TSE.1987.232894.

[28] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *Proceedings of the 2Nd Conference on Hot Topics in System Dependability - Volume 2*, HOTDEP'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251014.1251019.

[29] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043583. URL http://doi.acm.org/10.1145/2043556.2043583.

[30] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741976. URL http://doi.acm.org/10.1145/2741948.2741976.

[31] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: Practical accountability for distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):175–188, Oct. 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294279. URL http://doi.acm.org/10.1145/1323293.1294279.

[32] S. Duan, K. Levitt, H. Meling, S. Peisert, and H. Zhang. Byzid: Byzantine fault tolerance from intrusion detection. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, SRDS '14, pages 253–264, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5584-8. doi: 10.1109/SRDS.2014.28. URL http://dx.doi.org/10.1109/SRDS.2014.28.

[33] G. Latif-Shabgahi. A novel algorithm for weighted average voting used in fault tolerant computing systems. *Microprocessors and Microsystems*, 28(7):357–361, 2004.

[34] F. B. Schneider. Understanding protocols for byzantine clock synchronization. Technical report, Ithaca, NY, USA, 1987.

[35] H2. H2 Database Engine. http://www.h2database.com. Accessed: 2017-04-14.

[36] InfluxDB. InfluxData (InfluxDB) — Time Series Database Monitoring & Analytics. https://www.influxdata.com/. Accessed: 2017-04-14.

[37] KairosDB. KairosDB, Fast Time Series Database on Cassandra. http://kairosdb.github.io/. Accessed: 2017-04-14.

[38] J. a. Sousa and A. Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 37–48, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4671-1. doi: 10.1109/EDCC.2012.32. URL http://dx.doi.org/10.1109/EDCC.2012.32.

[39] DigitalOcean. DigitalOcean: Cloud computing designed for developers. https://www.digitalocean.com/. Accessed: 2017-06-02.

[40] M. Bravo, L. Rodrigues, and P. Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, Eu-

roSys '17, pages 111–126, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064210. URL http://doi.acm.org/10.1145/3064176.3064210.