Byzantine Fault Tolerant Monitoring of Distributed Systems

(extended abstract of the MSc dissertation)

Bernardo Palma
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***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.**

I. Introduction

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.

With this work, we present a MonS that is autonomous, robust and flexible, allowing the creation and management of different types of sensors, capable grouping data from various sources to export a coherent view of the main system, even in the presence of Byzantine faults. The remainder of this abstract is structured as follows: in Section II we present part of the context that supports and motivates our work, focusing on the monitoring strategies of adaptive systems already introduced. In Section III we present the system model and the assumptions made and taken into consideration during our work. In Section IV we present the general view of the system detailing its architecture, interfaces and main functionalities as well as the design choices made. In Section V we present the results on the evaluation of our work. Lastly, conclusions are presented in Section VI.

II. Related Work

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 approach the different solutions that have been introduced in said systems.

A. 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, a PANIC process is started and the next protocol in line is put in effect. Returning back when a quarantine period has passed.

B. 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.

C. 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.

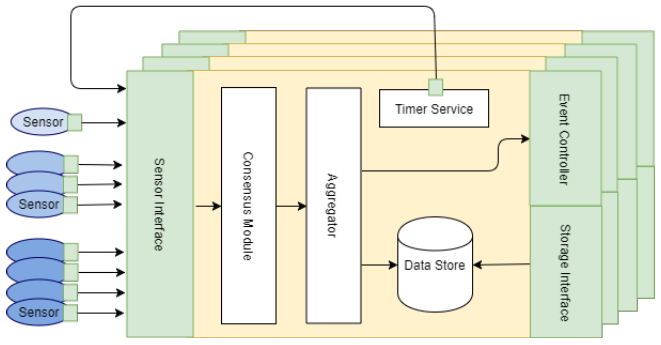

Figure 1. System Architecture

III. 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 Adaptation Manager (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.

IV. Architecture

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 system is presented in Fig. 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.

A. 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 I.

The data collected from the MS is passively sent to 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 need to wait for a response from the aggregator as they do not perform actions upon the system. This restriction is intentional, seeing as in order for the adaptation to be efficient, knowledge about the MS strategies is required, and so we believe they should be systematized and coordinated by a specific service.

Listing 1. Classe PeriodicSensor

```
package argus.sensors;
...
public abstract class PeriodicSensor extends BaseSensor{
public PeriodicSensor(Integer id, String type, PrivateKey
    pKey){...}
public abstract SignedMessage collectValue()}
```

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.

1) Extensibility: The managed system's goals may be subject to change, introducing new needs and consequently new possible adaptations. This may imply a 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 basic system startup.

For deployment we consider 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 API presents two ways of deploying the sensors, respectively, as a concurrent execution thread along with the MS or as a separate process launched either on the same machine as the MS or not. These were made to work with the sensor development interfaces also provided by the API. Namely, we offer an abstract class to create periodic sensors, shown in Listing 1 and one to store reactive sensor's information since as stated, the events monitored can range from many types.

B. 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 replication library called Bft-SMaRt[10], that is already capable of tolerating Byzantine

Table I
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

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, e.g 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.

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, and not only that but also allowing 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 then 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, for example, by removing the f lowest and highest readings and averaging the resulting values. In order to add more aggregation functions into the aggregator, an interface is provided. 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.

Listing 2. Example of an aggregation config file

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

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 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:

- The properties from the sensors may be defined in a file called *aggregation.config* (an example is provided in Listing 2) 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.
- 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 IV-A1, 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.

C. 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.

The prototype of this component is implemented as a relational database embedded in the system using H2[11]. For a more concrete/complete implementation the use of a proper time-series database such as InfluxDB[12] or KairosDB[13] would be beneficial as it would allow a broader and more complex manipulation of data natively.

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).

D. 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 IV-A. 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 V.

1) Post-Consensus Accumulation: PosC: 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 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 this section. A visual representation of the general message pattern for PosC is presented in Figure 2 (the internal message exchange of Bft-SMaRT is omitted[14]).

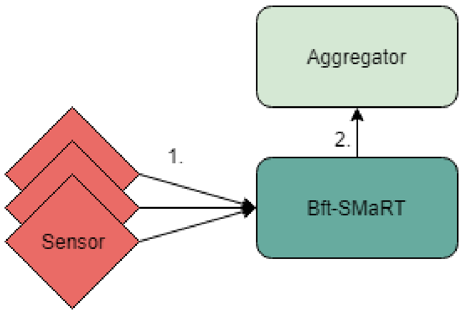

Figure 2. Message pattern for PosC and Integrated-PreC, S represents a sensor and its replicas.

2) Pre-Consensus Accumulation: 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, and thus the consensus is finally run. This implementation choice adds an extra communication step to the protocol when compared to the one presented previously, as shown in Figure 3. After the set of values is ordered, it is delivered back to the aggregator to be aggregated, 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;

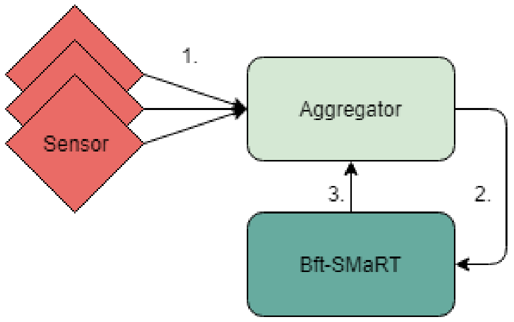

Figure 3. Message pattern for Total-PreC and Disperse-PreC.

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 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.

3) 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. 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 2, but does away with the need to run a consensus for every value received, which is relevant for replicated sensors.

E. Timer Service

A simple timer service is also provided. This service can be useful to notify, for example, an adaptation manager that a certain grace period has elapsed or that its time to reevaluate its policies. That being said, this notification only occurs if the timer is triggered in a majority of replicas of the system, passing through its consensus. The timers work as clients of the replicated state machine, using an internal client to the system. The events are sent as they are triggered, and as it happens with the sensors, the values pass through the aggregator where an accumulation occurs. Finally when the quorum is reached, the event can be delivered. Two types of timers are provide by the service: Periodic Timers (that, as the name implies, are triggered repeatedly with a rate defined in the moment its registered) and timers that are triggered once, with a delay defined at registration.

F. Event Controller

Although they are also stored in the data store, events are a type of data that in general require immediate reaction. As such, upon its occurrence the system feed by our own may require a particular action to be triggered. In order to provide this feature, our system offers the possibility of registering handlers associated to certain events. When the event is captured by the sensors and properly aggregated, the respective handler is called allowing, for example, simply informing the other system about its occurrence.

V. Evaluation

The evaluation is divided into two parts, and it seeks to answer the two following questions: How easy is it to extend the monitoring infrastructure of our system using the provided API and How do the performances from the different implementations presented in IV-D compare to one another, and more concretely does performing accumulation of values before the consensus yield any advantages?

The answer to the first question is omitted due to size constraints and its deferred to the accompanying thesis. For the second question the answer is presented ahead.

A. Evaluation Setup

The system and sensors used in the evaluation were hosted by Digital Ocean[15]. 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 II. The notable difference in specification is not with the intention of running a sensor more powerful than the system, but several smaller ones in a single environment mitigating the occurrence bottlenecks.

Table II
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.

B. 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 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), 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. 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.

1) 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, but the values are all 50ms except the link between sensor and leader which is the same.

2) 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 [16]. 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.

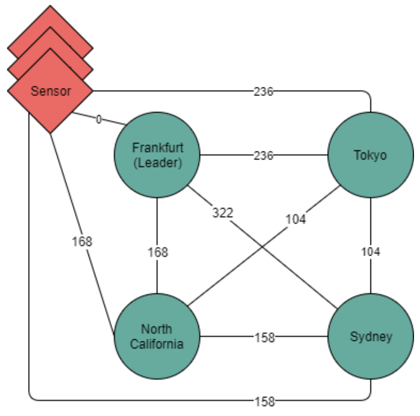

Figure 4. Real World Latency Scenario Topology. Values of latency presented as ms.

C. 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 we non-replicated sensors as stated before. The results are presented in Figure 5. 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.

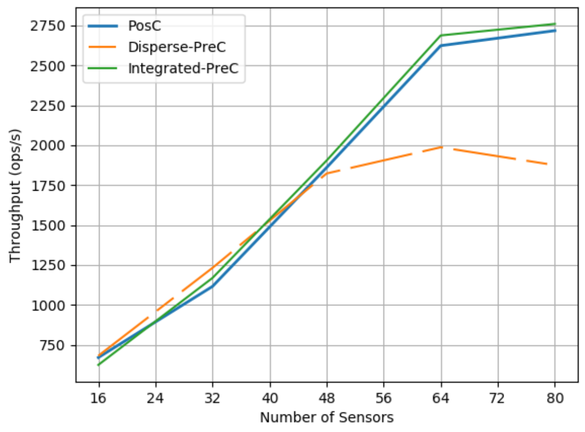

Figure 5. Performance results with non-replicated sensors in a latency free scenario.

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.

D. 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 6, 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.

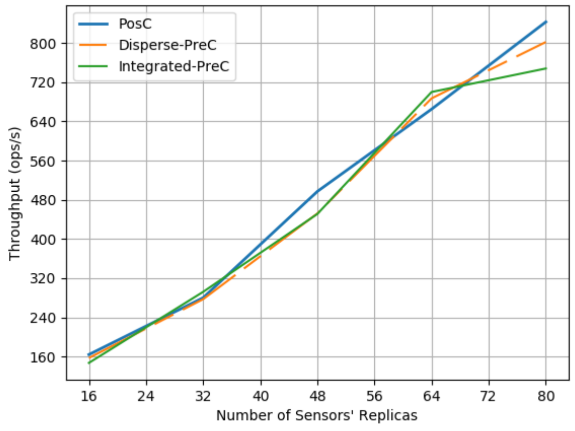

Figure 6. Performance results with replicated sensors in a latency free scenario.

As shown in Figure 7, 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 lesser throughput. 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 8, 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 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

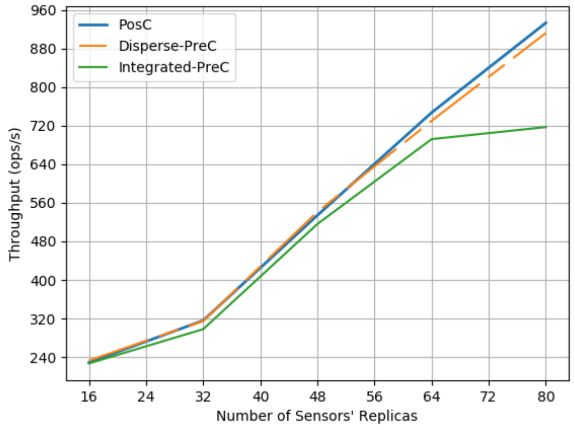

Figure 7. Performance results with replicated sensors in a flat latency scenario.

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.

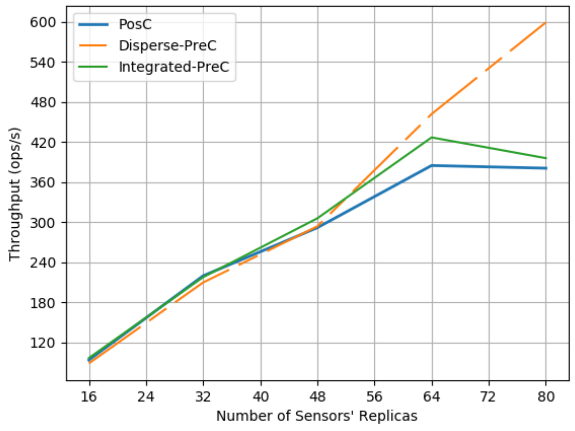

Figure 8. Performance results with replicated sensors in a real world latency scenario.

E. 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 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.

VI. Conclusions

With our work, 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.

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..

References

[1] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010. [Online]. Available: 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*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–168. [Online]. Available: 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*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. [Online]. Available: 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*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 59–74. [Online]. Available: 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*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 177–190. [Online]. Available: 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*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 363–376. [Online]. Available: 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.*, vol. 8, no. 4, pp. 564–577, Jul. 2011. [Online]. Available: 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*, May 2015, pp. 904–913.

[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, Sep. 2016.

[10] 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*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 355–362. [Online]. Available: http://dx.doi.org/10.1109/DSN.2014.43

[11] "H2 Database Engine," http://www.h2database.com, accessed: 2017-04-14.

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

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

[14] 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*, ser. EDCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 37–48. [Online]. Available: http://dx.doi.org/10.1109/EDCC.2012.32

[15] "DigitalOcean: Cloud computing designed for developers," https://www.digitalocean.com/, accessed: 2017-06-02.

[16] 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*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 111–126. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064210