UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

FIT-BROKER: DELIVERING A RELIABLE SERVICE FOR EVENT DISSEMINATION

Igor Daniel Cristina Antunes

DISSERTATION

MASTER ON INFORMATION SECURITY

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

LISBOA

UNIVERSIDADE DE LISBOA

FIT-BROKER: DELIVERING A RELIABLE SERVICE FOR EVENT DISSEMINATION

Igor Daniel Cristina Antunes

DISSERTAÇÃO

Trabalho orientado pelo Professor Doutor António Casimiro Ferreira da Costa
e co-orientado pelo Mestre Diego Luís Kreutz

MASTER ON INFORMATION SECURITY

2013

Thanks

First of all, I would like to thank my advisor António Casimiro and co-advisor Diego Kreutz, without whom I would not have been able to fulfil this work.

The next thank you goes to my girlfriend Leonor Pinto, my mother Narcisa Cristina and to all my family for their support and patience.

Also, I would like to thank my friends and colleagues with special thanks to Jaime Mota Vaz, Pedro Pereira, Pedro Nóbrega Costa and João Varino Alves for their help. Additionally, I leave special thanks to all my colleagues from lab. 25 for their support and help in the difficult moments.

Resumo

Os serviços de nuvem (*Cloud*) estão a assumir um papel cada vez mais importante no mundo de fornecimento de serviços. Estes serviços variam desde a oferta de simples ferramentas de trabalho até a disponibilização de infra-estruturas remotas de computação. Como tal, a correcta monitorização das infraestruturas de nuvem assume um papel vital de forma a garantir disponibilidade e o cumprimento de acordos de nivel de serviço.

Existem alguns estudos recentes que mostram que este tipo de infra-estruturas não se encontra preparada para enfrentar atuais e futuros problemas de segurança que podem ocorrer. Parte deste problema advém do facto de as ferramentas de monitorização serem centralizadas e de apenas suportarem alguns tipos de falhas.

De forma a tornar os sistemas de monitorização mais resilientes, esta dissertação propõe uma solução para aumentar a confiabilidade no transporte de informação entre os seus varios pontos. Trata-se de uma framework adaptável e resiliente de dessiminação de eventos baseada no paradigma de publicador-subscritor. Esta oferece múltiplos níveis de risilencia e qualidades de serviço que podem ser combinados para oferecer uma qualidade de serviço e de proteção adequada as necessidades de cada sistema.

Este documento descreve a arquitectura da framework bem como todo seu funcionamento interno e interfaces oferecidas. Este documento descreve ainda um conjunto de testes realizados de forma a avaliar a performance da framework em vários cenários distinctos.

Palavras-chave: Faltas Byzantinas, Monitorização de sistemas, Sistema produtor-consumidor

Abstract

Cloud services are assuming a greater role in the world of service providing. These services can range from the simple working tool to a complete remote computing infrastructure. As such, the correct monitoring of this type of infrastructures represents a key requirement to ensure availability and the fulfilment of the service level agreements.

Recent studies show that these infrastructures are not prepared to face some current and future security issues. Part of these problems resides in the fact that current monitoring tools are centralized and are only prepared to deal with some types of faults.

In order to increase the resilience of monitoring systems, this dissertation proposes a framework capable of increasing the reliability of the transport of information between their many peers. It is a adaptable and resilient framework for event dissemination based on the publisher-subscriber paradigm. The framework offers multiple levels of resilience and quality of services that can be combined to meet the necessities of quality of service and protection of each system.

This document describes the architecture, internal mechanism and interfaces of the framework. Also, we describe a series of tests that where used to evaluate the performance of the framework in different scenarios.

Keywords: Byzatine Faults, System monitoring, Publish-subscribe systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A typical cloud infrastructure is composed of computing, storage and network hardware that are interconnected. Normally, on top of the hardware we have complex stacks of software. With this, it is possible to offer various types of services to remote clients, according to to three main models, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). For cloud providers, monitoring these infrastructures is important in order to diagnose possible problems and help to maintain high indices of availability and security.

A monitoring system is composed of agents and probes that are deployed on the infrastructure and collect information. The information is then sent to the backend that process it and displays relevant events. The backend can vary from viewing consoles to Security Information and Event Management (SIEM) that process the information and launches alerts if it detects security risks.

The monitoring systems can suffer problems if the channels that are used to transport the information from the infrastructures to the backend are not reliable an can suffer faults. Thus leaving the system blind to any possible problem that may occur in the infrastructure.

This work proposes a framework that can increase the resilience of monitoring systems. We provide a fault and intrusion tolerant event broker (FIT-Broker) based on the publish-subscriber paradigm. The FIT-Broker offers means to replicate the transport channels thus increasing their reliability. Also, by being based on the publish-subscribe paradigm it provides ways to transport the same information to multiple backends.

1.1 Motivation

Currently, many companies are relying on cloud services and expect *100%* of uptime. To monitor the correct and continuous operation of their systems, cloud providers rely on their monitoring systems. This is due to the fact that unscheduled downtime can translate into losses in terms of clients and revenue.

Monitoring systems collect information from the infrastructure using probes or agents. This information is then processed and stored. The stored information is then correlated and analysed, and if availability or security problems are detected, alerts are raised.

The monitoring systems typically follow a centralized architecture, thus they can become single points of failure or attack. This means that if it becomes compromised it will affect the overall computing infrastructure in terms of monitoring and control. One existing solution that follows this centralized approach is ArcSight Enterprise Security Manager (ESM). It collects security related data and log information from various devices that are sent to a backend. In ESM, the backend is composed of a centralized correlation engine that analyses events based on their context and raises alarms when it detects critical situations. The alarms are presented in viewing consoles that can show various perspectives of the same system or different systems. These consoles can also become a single point of failure or attack.

Additionally, monitoring systems can fail if the channels that are used to transport information between the multiple peers can suffer faults. If channels fail, the system can face problems of starvation, meaning that system monitors will not be able to presents the status of the infrastructure.

To surpass the above mentioned problems, we can resort to replication. By replicating the communication channels, we are increasing their reliability by decreasing the probability that all of them fail simultaneously. Also, by replicating the communication channels, it becomes possible to send the same information to multiple backends. This allows the replication of the monitoring consoles, thus ensuring a correct monitoring of the systems even if one of the consoles fails or is attacked and displays wrong information.

To increase the reliability of channels, a middleware that is capable of transporting the same information to multiple backends is required. It should be capable of ensuring a trusted and reliable event propagation between all the probes and all the backends. To avoid creating single points of attack or failure, the middleware itself should be replicated. Also, by replicating the middleware we can provide various quality of services and protection that can be combined to better suit the needs of the monitoring systems. In controlled environments where only crash faults are predicted and thus assumed in the fault model, only two replicas may be required. While in environments where arbitrary faults may occur, a greater number of replicas are necessary.

1.2 Objectives

The main objective of this work is to implement a fault and intrusion tolerant framework based on the publish-subscribe paradigm, and its evaluation under different situations. The framework is called FIT-Broker which stands for Fault and Intrusion Tolerant Broker [16].

Achieving this objective is challenging because the FIT-Broker must provide a number of properties, such as ordering of events, the ability to handle events according to different fault-tolerance requirements for event channels, and also must provide some measures to achieve a reasonable performance.

Therefore, in addition to the main objective, this work also aims at:

- Implementing the necessary solutions to deal with both crash and Byzantine fault tolerance requirements for the event broker;
- Implementing solutions to allow event batching;
- Fine tuning the developed code in order to fulfill throughput requirements, as best as possible;
- Develop a testing platform and perform the necessary experiments to assess the correct behavior of the implemented FIT-Broker, as well as to assess the achievable performance.

A final objective of the work is to develop a simple demonstration scenario, in which the FIT-Broker is used in connection to some publishers and some subscribers, illustrating the behavior of the broker under a set of situations, including failure situations

1.3 Context of the work

This work is performed in the context of the Trustworthy and Resilient Operations in a Network Environment (TRONE) project. The TRONE project has the objective of enhancing network quality of service (QoS) and quality of protection (QoP), operational efficiency and agility, in the context of accidental and malicious operational faults of expected increasing severity [16, 27].

The project plans to achieve its objectives by investigating paradigms and creating mechanisms to achieve a trustworthy network operations environments through: *a)* developing mechanisms to ensure real-time operational security and dependability through on-line fault and failure diagnosis, detection and prevention, recovery and adaptation; *b)* developing a midleware to increase the reliability of the network management and monitoring infrastructure itself under situations like overload or attack; c) develop prototypes to demonstrate the practicality of the mechanism and techniques that were created.

The work presented in this document contributes to the objectives of TRONE by providing the FIT-Broker middleware to increase the reliability of monitoring infrastructures.

1.4 Document structure

This document is structured as follows: chapter 2 contains the related work of this project. Next, chapter 3 presents the architecture and system model of the framework. Chapter 4 the protocols that are used and our implementation of the proposed architecture, followed, in chapter 5.2, by the evaluation of the implementation and the presentation of a small application that is used as proof of concept. In the last chapter, we make a brief conclusion of the work and talk about possible future work.

Chapter 2

Related Work and Background

Our objective is to provide a solution for improved system monitoring by developing a Fault and Intrusion Tolerant (FIT) event broker that builds on the publish-subscribe paradigm for event dissemination and on fault tolerance techniques for reliability. As such, the first section will briefly describe monitoring systems. Following, the next section introduces the publish-subscribe paradigm. After that we describe the fault models. Lastly, section four and five present basic concepts about Byzantine fault tolerance and intrusion tolerance, respectively.

2.1 System monitoring

System monitoring is a key aspect of any computer infrastructure. It is required as a basic building block for improving the system behaviour and assuring that any potential problems that may occur can be detected by analysing the information collected by the monitoring system. As such, system monitoring represents a crucial task that should not be neglected.

According to [38], system monitoring can be defined as the process of obtaining information of a computer system and processing it. One can use different categories of information to monitor a system, such as:

- Status information
- Configuration information
- Usage and performance statistics
- Error information
- Topology information

- Security information

Once collected, the information can pass through a sequence of processes like cleaning raw-information, compacting it into more manageable amounts, and preparing reports to be viewed by the system administrators.

A typical monitoring system is composed of different modules and an execution work-flow, as illustrated in Figure 2.1 [38]. There are five main modules. Each of them has a particular purpose and is independent of the other modules, as in a loosely coupled architecture. Based on their decoupled nature, the modules can be distributed across different machines in order not to overload a single machine, for instance. Nevertheless, the data follows the represented flow.

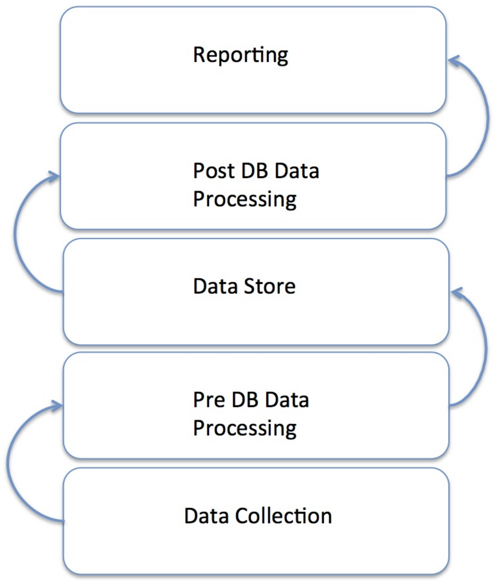

Figure 2.1: Common modules and workflow of a monitoring system

The data collection module is responsible for collecting information from the infrastructure. It is composed of a set of agents and probes that collect raw information from software and devices, which is then forwarded to the next module. The data collection can either be passive or active.

In passive mode, probes collect information that is generated from normal operation of the monitored system (ex: log files). Therefore, it doesn't impose any extra load on the system, besides augmenting the network traffic. In contrast, active monitoring generates extra workload on the system due to the synthetic operations used to asses its status. For example, if we are monitoring a web server, an operation could be the request of a web page, while the response time could be used to evaluate its status.

The Pre DB data processing module is responsible for reducing the volume of information that needs to be sent to the upstream monitoring modules. It does this by cutting redundant, erroneous or incomplete data and converting it to a single format to be stored in some databeses.

The output of the Pre DB module is then stored in the data store module. This module is normally composed of Commercial Of The Shelf (COTS) databases that can have different designs (rolling, load-balanced, hierarchical federation, partitioned) and are chosen according to the volume of information that is expected. It is possible to have multiple databases, like one for each type of information (ex: one for security information and another for usage and performance statistics) or to distribute the data load.

The Post Database Data Processing module has the intelligence of the monitoring system. It is composed of algorithms that analyse collected information and decide what to send to the next module to be presented. For instance, it correlates information from different sources, and uses patterns to determine specific alarm situations.

Lastly, we have the Reporting module which is responsible by presenting the information to system administrators. It is normally composed of viewing consoles and monitors that should present the system information in an effective and simple way.

Another type of monitoring tools that follow the same architecture are the cloud monitoring tools. They must be able to monitor the systems running in virtual machines, monitor the virtual machines, monitor the hardware where the virtual machines run, and deal with big amounts of information.

There are several cloud monitoring tools [27, 7, 42, 1, 39, 22, 24, 35]. Each of them was conceived with different purposes and objectives. But all of them have the common goal of providing run time information about the cloud infrastructure. Many companies rely on these tools to monitor their systems.

2.2 Publish-Subscribe Systems

Publish-subscribe systems are a suitable alternative when we have two types of clients, producers and consumers. Producers can be seen as publishers, while consumers become subscribers. A message routing middleware (also know as message broker) is used to route messages between producers and consumers. It receives events generated by publishers and makes them available to all interested subscribers. These systems promote the decoupling of clients. Consequently, publishers and subscribers do not need to know each other.

Publish-subscribe systems can be divided in two architectures: centralized and decentralized. As represented in Figure 2.2, centralized architectures follow the client-server model. Publishers and subscribers are the clients that connect to message brokers. These

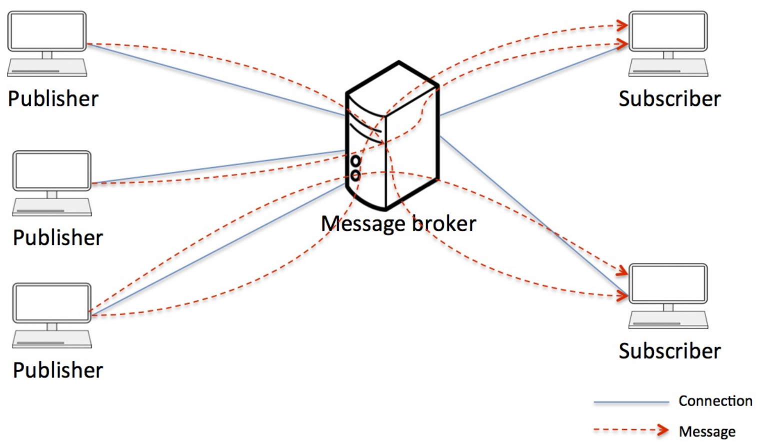

Figure 2.2: Centralized publish-subscribe system

brokers are responsible for routing messages produced by the publishers, to interested subscribers.

This model can work well for monitoring systems. The message broker acts as intermediary between peers of the system. Probes and agents become publishers, while viewing consoles become subscribers.

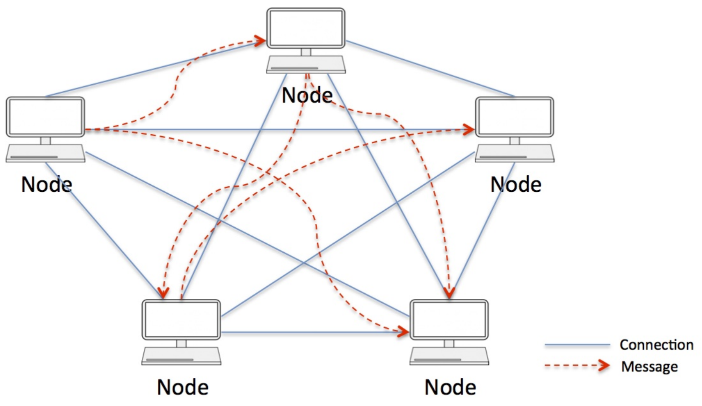

Figure 2.3: Fully connected peer-to-peer network

Decentralized architectures are composed of nodes, also known as peers, that have the same functionality in the system. In this model there are no message brokers, peers announce themselves as publishers for one or more types of messages, and interested peers can become subscribers. The most common example of these systems are the peer-to-peer (P2P) networks [31]. They can be divided in two fully connected and hierarchical overlay networks. In fully connected P2P networks, as depicted in Figure 2.3, each peer is connected or knows all the other peers. Figure 2.4 exemplifies a hierarchical overlay network, in this architecture we have peers that connect to super peers (chosen according to their connection speed or availability). A super peer acts as a message broker to its peers. It connects itself to other super peers and acts in a similar manner to peers in fully connected networks.

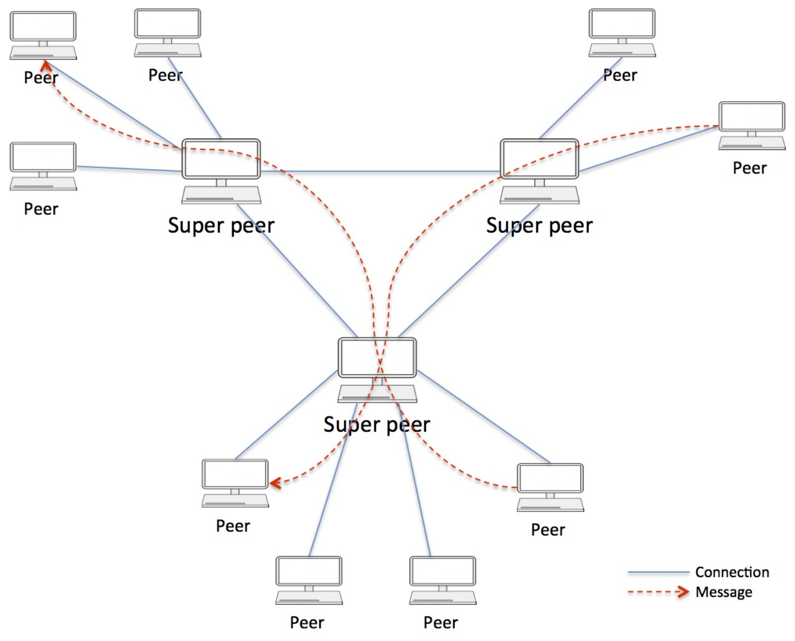

Figure 2.4: Hierarchical overlay network

There are researches that compare publish-subscribe systems based on the decoupling of clients [10, 21]. These systems can route messages from publishers to subscribers based on several aspects like topic, content or type. The most common use is the content routing, where subscribers can specify the type of information they wish to receive. For example, a subscriber wants to receive information about stocks, but only the ones that went up in the last 24 hours, meaning that the information about stocks either has a time to live or a minimum value to be valid. All the information that does not fulfil the requirements is discarded by the message broker.

Routing algorithms for publish-subscribe systems are the subject of several researches [2, 11, 13]. Most of these algorithms are not reliable, in the sense that they do not tolerate arbitrary faults. To our knowledge only [32] is capable of tolerating arbitrary failures. However, this algorithm was designed for blind-auctions (where clients make bids and do not know the value of the highest bid) and faults are only detected after the auction ends. This leaves a narrow space for adaptation to other purposes such as content based message routing.

2.3 Dependability

Companies that depend on computer systems expect to have close to 100% of uptime. However, during the lifetime of computing systems, faults may occur.

Faults can happen because computer systems are complex artefacts, composed of several hardware and software components that interact with each other, and sometimes unpredictable interactions occur that cause a fault. A component fault can lead to an error, which in turn can cause a system failure. There are several classes of faults: omissive class, assertive class or arbitrary class (also known as Byzantine faults) [37].

When a system component stops working, sporadically misses an operation or performs it at a late time, we are in the presence of an omissive fault. These faults are said to be in the time domain.

If components are affected by faults in the value domain, we have an assertive fault. They can be divided in two groups: semantic and syntactic faults. A semantic fault occurs when a component performs an operation within the correct context but with an incorrect result. For example the sum of two numbers is affected by a fault and the result is wrong although it could be a correct result in the context. Syntatic faults occurs when a component performs an out of context operation. If the sum of two numbers is a letter then we are in the presence of an syntatic fault.

Arbitrary faults occur when it is not possible to say if a fault belongs to the time or the value domain as depicted in Figure 2.5. This class of faults is used when system designers can not make any prediction about the behaviour of faulty components. For example, if a sequence of accidental events occurs that leads to an unpredictable state, or when a malicious component (intruder) tries to take down the system.

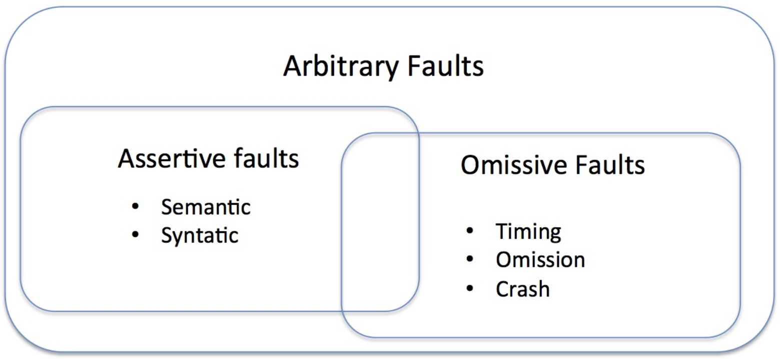

Figure 2.5: Fault classes

In this project, we are specifically interested in dealing with omissive and arbitrary faults. Both classes are described in the next section.

2.3.1 Omissive faults

Omissive faults can happen if components either fail by stopping, by omitting some operation or by performing it with some degree of lateness.

Figure 2.5 shows that omissive faults can have three different but similar natures. In the case of timing faults, components perform an operation at a late time. The delay between the time that an operation is supposed to take place and the actual time where it takes place is called the lateness degree. If the degree is infinite, we have an omission fault, meaning that the component did not and will not perform the operation. This can be seen has a particular case of the timing faults. A crash fault is when components stop working. If we consider that the sucessive number of omission faults is the omission degree, we can say that a crash fault is a particular case of the omission fault where the omission degree is infinite.

To cope with faults of the omissive class, system designers should assume an upper bound f on the number of faults that can occur. In this case, the system is going to crash or start to malfunction only after f faults. To support f simultaneous faults each component of the system has a *resilience degree* of $n \geq f + 1$. This will ensure that even if there are f successive faults, there is still one component performing the correct operation. By using this redundancy, omissive faults can thus be masked.

2.3.2 Arbitrary and assertive faults

When no assumptions can be made about the type of faults that the components will suffer, we assume arbitrary faults, also known has Byzantine faults [19]. This means that components can suffer any kind of faults, including those of malicious nature.

In this fault model, it makes sense to consider that any deviation from the protocol is a fault. As such, assertive faults fall under it, since they state that components don't behave as expected. In this model, a malicious component (taken over by an intruder) can be used to take down the entire system or to induce it to an erroneous state. This may include giving different or out of context results to the same operation (assertive faults), performing omissive faults, or both types of faults as shown if Figure 2.5.

The common approach to deal with faults of this nature, is to assume an upper bound f on the number faults. After it is crossed, no assumptions about the behaviour of the system can be done.

Given that in this model we make no assumptions about the behaviour of the system, a higher resilience degree is necessary to ensure that components follow their specified protocol. This means that having $n \geq f + 1$ isn't enough. The *resilience degree* for the arbitrary and assertive fault model must be $n \geq 2f + 1$. This higher number of redundancy comes from the fact that if we have an assertive fault we can't say which component is the faulty one. In order to get a correct result and detect a faulty component, we need to reach a quorum of *f+1* equal results, and so a total of $f + 1$ (correct components) plus f (possibly incorrect component) is needed.

2.4 Fault and intrusion tolerance

Distributed systems that are designed to be fault and intrusion tolerant must be capable of performing correctly even in the presence of faults and intrusions.

By considering Byzantine fault tolerance techniques, it is possible to mask all faults. Therefore, even if there is an intrusion in the system, it can be masked (provided that no more than f replicas are compromised).

Opposed to intrusion prevention which aims to avoid the occurrence of faults (i.e. making $f = \emptyset$), intrusion tolerance allows the attacker to breach the system. These breaches can be seen as a fault [36]. When assuming a Byzantine fault model, solutions similar to the described in Section 2.5 are employed. With these, the system will work properly if no more than f simultaneous faults occur. To ensure the continuous uninterrupted operation of the systems, the use of replication should be complemented with recovery and diversity mechanism [3].

2.4.1 Recovery

For ensuring that no more than f faults occur, the detection and recovery of faulty replicas is critical.

By using the State Machine Replication (SMR) paradigm, it is possible to construct replicas that can recover from faults. This can be done because by using this paradigm, each replica has a state that is equal across all replicas. When a replica fails, we can reboot it and transfer the current state of the system from other replicas. The state must be requested and transferred from correct replicas.

To detect faulty replicas, we can implement a failure detector in all replicas of the system [14]. The detectors will monitor the other replicas and, if a fault is detected, the faulty replica will be notified and isolated. Once isolated, it can be recovered.

2.4.2 Diversity

Tolerating intrusions does not mean that they shouldn't be prevented. To avoid having the same vulnerabilities in all the replicas of the system, diversity should be introduced.

It is possible to introduce diversity in several different places like: *a)* operating systems; *b)* virtual machines; *c)* code; *d)* memory stack. Each of these, can make the job of an attacker more difficult.

When recovering a faulty replica, a system with diverse components should be used to replace the corrupted system.

2.5 Byzantine fault tolerance

Fault tolerance is the ability to provide a correct service even in the presence of faults. Systems capable of tolerating omissive faults are usually composed of *f+1* replicas and are able to tolerate up to f faults. These systems can use active or passive replication. In active replication, all replicas execute all the commands and maintain an equal state. In passive replication, only one replica is active at each moment and is responsible for maintaining the system sate. When this replica fails, a stand-by replica must take its place by first re-executing the commands needed to get to the state of the failed replica. But, what happens in the case of Byzantine faults?

The Byzantine fault model assumes that faults can have accidental or malicious nature. Faults like state corruption, generation of inconsistent outputs or intentionally producing incorrect outputs can occur. Byzantine Fault Tolerance (BFT) is the ability to provide a correct service in the presence of such faults. When employing BFT techniques, the systems are capable of tolerating and mask up to f Byzantine faults. Systems that employ BFT techniques use active replication.

If state machine replication is used, the replicas have to communicate with each other to reach a consensus on the operation to perform. For this reason, the resilience degree must increase to $n \geq 3f + 1$. This is driven by the fact that f of the replicas may be compromised making it impossible to reach consensus on the operation with only *2f+1* replicas. The definition of the problem is presented in [19] and states that it is not possible to reach consensus with three participants if one tells different things to the other two. To solve problems of this nature, a fourth participant is required.

Works aimed on reducing the resilience degree required to solve these problems have been conducted. By separating the agreement protocol from the execution, it was proved that it is possible to use *2f+1* replicas for execution, however the agreement still requires *3f+1* replicas [41]. By using virtualization, the authors of [40] demonstrate that only *f+1* active replicas are required to mask up to f Byzantine faults.

2.5.1 State machine replication

State Machine Replication (SMR) uses the concept of State Machine (SM) in replicated system. The basic idea of a SM is that we have a machine or a computer process that has a state and only executes deterministic operations [17]. These operations are performed according to the input, and after their execution, the state is updated. The progress of the state machine can be seen as the evolution of its states, which in turn is dictated by the input.

SMR is the application of the SM concept in a distributed fashion, which means that we have several replicas of the same SM [30]. To maintain equal states, it is important to ensure that all replicas receive the same input in the same order. Achieving this is no simple task, since it implies that all replicas reach an agreement on the input they receive at each step.

To ensure that replicas reach an agreement on the order of the message they receive, message ordering protocols are employed. These will execute a series of operations after which, all correct replicas will have the same order of messages.

2.5.2 Message ordering

In systems composed of several communicating peers, it can occur that two peers will receive the same set of messages but in different orders. There are several protocols [9, 18, 20] to ensure that all peers receive the messages in the same order. Atomic broadcast [9] is a primitive which ensures that messages sent to a set of peers are delivered by all of them in the same order.

When assuming Byzantine faults, to achieve consensus on the order of messages, the replicas require a more complex set of protocols like the VP-Consensus [33] or the Paxos at war [43]. There are other protocols capable of doing this like the ones described in [29, 33]. All of them are leader-based.

In leader based protocols we have a leader, that is responsible for ordering the messages. Normally, the leader proposes an order to the messages, an all correct replicas (including the leader) will execute a series operations before acknowledging and accepting the order.

Chapter 3

FIT-Broker Architecture

In this chapter we present the Fault and Intrusion Tolerant event broker (FIT-Broker) architecture. We start with the requirements, followed by the system model and lastly the architecture.

3.1 Requirements

The FIT-Broker has non-functional and functional requirements. First we present the non-functional requirements followed by the functional requirements.

3.1.1 Non-functional requirements

Integration with existing systems is the first non-functional requirement. The FIT-Broker should be used by any system that wants to transport its events in a trustworthy way. It must provide a simple interface, that can be used by any system willing to use it for reliably transporting their information.

The second non-functional requirement is scalability. The service offered by the FIT-Broker must be scalable, meaning that it must be possible to increase the number of client of the system. By increasing the number of clients, the number of events that the system must process increases, thus the system must also be scalable in the number of operations it can process per second.

Availability is our third functional requirement. The FIT-Broker must provide a correct service 24x7, with a sufficiently high probability. This means that it must be complemented with recovery mechanisms. These mechanisms will allow the FIT-Broker to provide a correct service through its life time even if faults occur.

The fourth requirement is Quality of Protection (QoP). The FIT-Broker must provide two different QoP: *a)* crash fault tolerance (CFT); *b)* Byzantine fault tolerance (BFT). In CFT mode, events created by publishers will be sent to all FIT-Broker replicas. Subscribers will receive them from the fastest replica and discard all other copies of the event. In the BFT mode, publishers will send their events all FIT-Broker replicas. Subscriber waits for a minimum of *f+1* equal events before considering them as correct.

3.1.2 Functional requirements

The first functional requirement of the FIT-Broker is to offer clients an interface that allow them to invoke operations like publish or register on the FIT-Broker replicas. The interface must provide a connection ”bridge” between clients and replicas. This means that it must be able to translate a clients intention into a format that the FIT-Broker understands and vice-versa.

The second functional requirement is Quality of service (QoS) regarding event urgency, ordering and persistence. Events marked has urgent will be delivered first to the subscribers. In a channel with ordered events, subscribers receive the content in the same order, otherwise no assumptions can be made about the order of events. Persistence is assured to a certain degree by event queues that can vary from channel to channel an have an upper bound (different from channel to channel) that dictates for how long an event is valid. If the queue is full, events are discarded according to the system owner preferences.

Configurable channels that provide temporary storage for events is our third functional requirement. Each channel on the FIT-Broker has an unique TAG that identifies it and stores events published by publishers and removes them when they are retrieved by subscribers. Each channel can have multiple publishers and multiple subscribers. Clients will register to them according to their needs.

3.2 System Model

In this section we talk about the system model assumed for this project. We present the network model, followed by the synchrony model and the fault model.

3.2.1 Network model

A fully connected network that follows the TCP/IP model is assumed. With this assumption, all clients are capable of reaching all the event broker replicas. Clients and replicas may be located inside the same Local Area Network (LAN), in different Virtual LAN (VLANs) or across several different LANs.

When the peers of the system are in the same LAN, they all see each other. If peers are in different VLANs, there is at least one connection point between them like a switch or a gateway. In cases where we have clients and replicas spread across several different physical LANs that are interconnected via a public network (ex: Iternet), Virtual Private Networks (VPN) may be used. The use of VPN allows companies to reduce costs since they do not require the use of private lines. However and since VPN connections create a virtual network over a public network, they can pose a security and availability threat due to the fact that public networks may suffer overloads, attacks, instabilities or temporary unavailability.

3.2.2 Synchrony model

The timed asynchronous model [8] is assumed. In this model, each process has access to a physical clock with a bounded drift rate. This assumption enables the system to measure the passage of time and to detect performance or omission failures. The model allows the implementation of protocols that can resolve problems like membership, leader election and consensus in a pre-defined time interval. This is reasonable assumption since most computing systems have high-precision quartz clocks.

3.2.3 Fault model

Two distinct fault models are considered: *a)* crash fault model; *b)* Byzantine fault model. By using the crash model, the system can provide service in the presence of f faults, if at least there is one correct replica forwarding messages to the clients. When the Byzantine fault model is assumed, the system will be able to tolerate up to f Byzantine faults if there are enough correct replicas forwarding messages to be voted by clients.

By assuming these two fault models, we consider all possible failures that may affect the correct behaviour of the FIT-Broker. For example, systems can suffer omission faults or crash faults due to power failures or software and hardware problems. It may also happen, that systems behave erratically due to delays or corrupted data, either intentionally or accidentally inserted into the system.

3.3 System Architecture

This section presents the system architecture. It starts by giving a general view of the architecture and of the relations between the components. Next we present the components of the FIT-Broker and lastly we describe the data structures that are used to invoke operations.

3.3.1 General view

The FIT-Broker is based on the publish-subscribe paradigm and provides a reliable and trustworthy communication layer for transporting events from publishers to subscribers. It follows a client-server model were the event broker is the publish-subscribe server and publishers and subscribers are the clients.

As represented in Figure 3.1, the FIT-Broker is composed of several replicas that are based on the SMR concept. Each replica executes the same operations given the same input (they are deterministic, as required by the SMR approach). This allows the execution of the required protocols to provide the desired QoP. For clients, the fact that the FIT-Broker is replicated is transparent. This is achieved by using interfaces on the client side that provide the abstraction of a single FIT-Broker server.

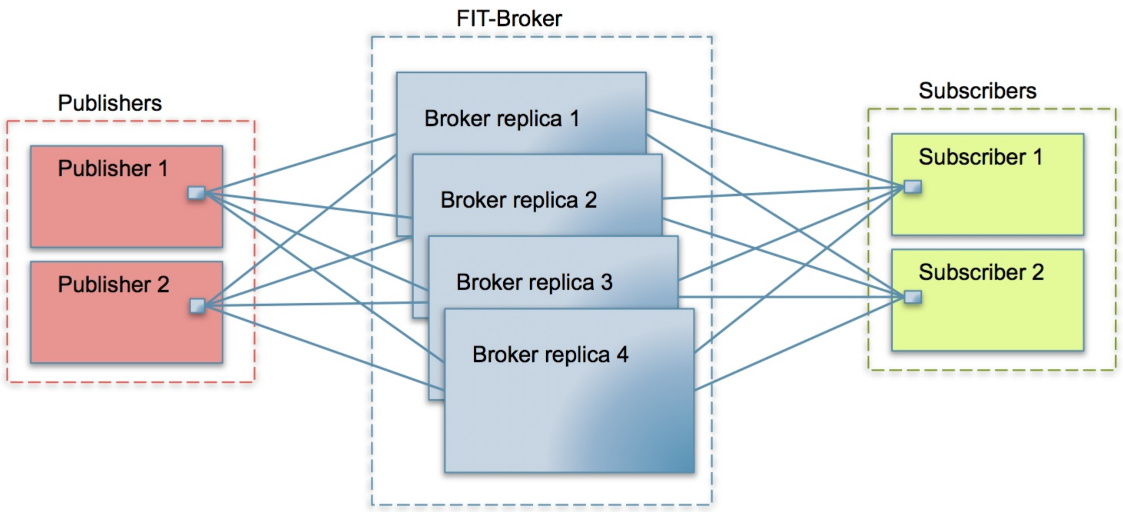

Figure 3.1: FIT-Broker

In figures 3.2 and 3.3 we have a representation of the process of a client interaction with the FIT-Broker. When a client wants to invoke an operation, it contacts the FIT-Broker interface (1) that uses a service proxy (2) to broadcasts a request that represents the client invocation to all replicas (3). When a replica receives a request, it makes use of a service proxy to verify the QoS and QoP requirements and perform the necessary operations that depending on the requirements, may or may not involve communication between the replicas (4). Once the service proxy concludes its operations, it delivers the request to the storage that is based on the SMR concept, containing the channels of the FIT-Broker and representing its state (5). A response is then prepared by the storage that is passed to the service replica (6) and sent back to the client (7). Once a reply is received, the service proxy verifies if it has enough conditions to attest that the reply is correct and when it does (8), delivers the reply to the client via the FIT-Broker interface (9).

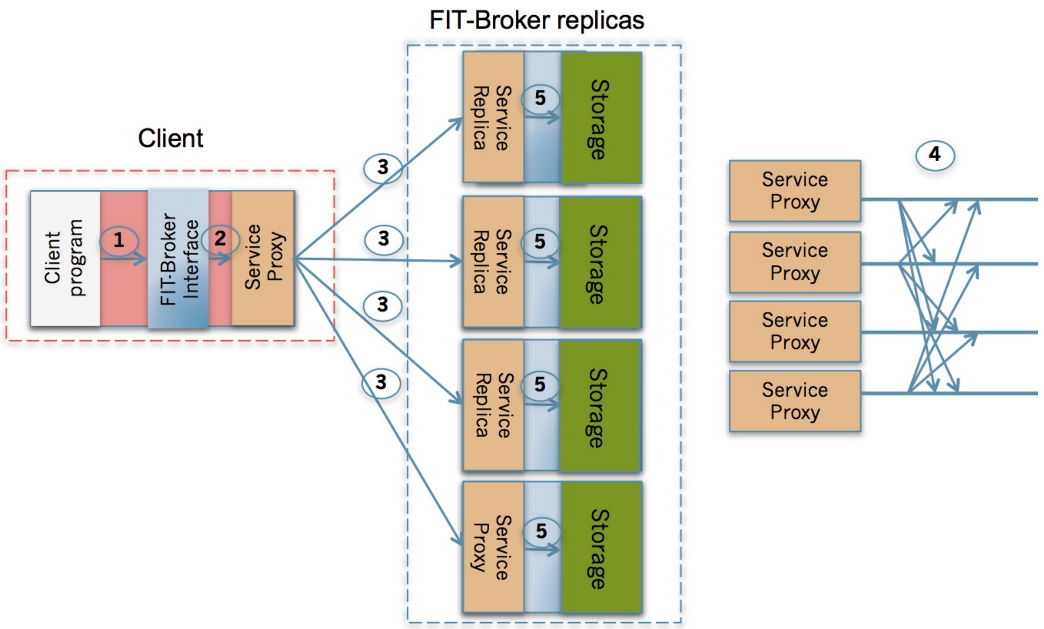

Figure 3.2: FIT-Broker interaction part 1

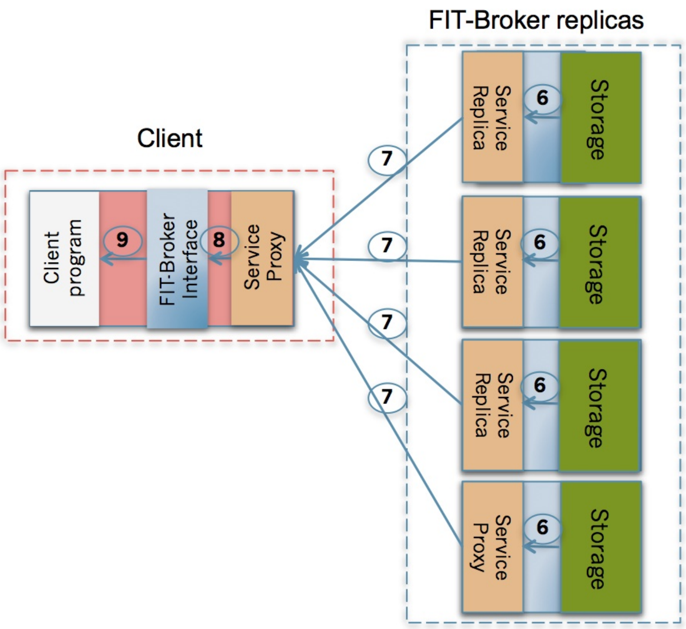

Figure 3.3: FIT-Broker interaction part 2

3.3.2 Functional components

The functional components are grouped into three abstract layers which are distributed across different nodes of the system. The representation of the FIT-Broker layers is presented in figure 3.4. Each layer has a specific function. Layer 1 defines the operations that the client can invoke. It is composed of interfaces that are used by clients to register, unregister, publish or consume events from the FIT-Broker. All operations that are invoked by the clients in this layer are transported to layer two and layer three where they are executed.

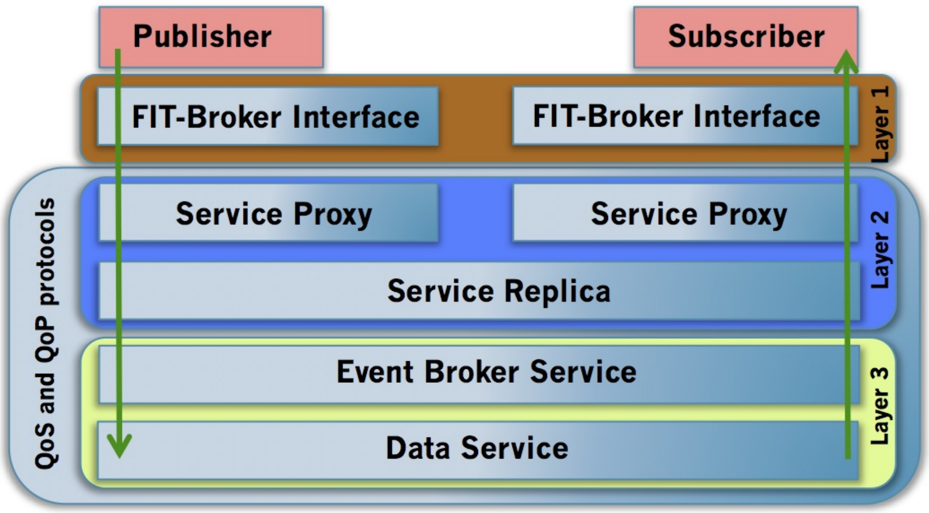

Figure 3.4: FIT-Broker service layers

Layer 2 is composed by the service proxy (client side) and the service replica (replica side) and implements all the communication mechanism required between clients and brokers. They are used to transport events to the FIT-Broker. Also, this layer is responsible for executing all the QoS and QoP protocols that are implemented in the FIT-Broker. This was the layer that was most time consuming along the realization of this project as it required the design, adaptation and implementation of ordering and fault tolerance protocols.

Layer 3 contains the event broker service and the data service. The first executes the operations defined in layer 1 while the second provides channel control functionalities such as event routing and managing.

The data service is composed by channels. Each channel contains queues to where events are published. Channels are created by replicas using configuration properties that can vary from channel to channel. These configuration properties are described in Table 3.1. The table is composed by the property name which indicates the name that must be used inside the configuration, the possible values that each property can assume, and a brief description of its meaning. Each replica has a copy of these properties, and thus generates the same channels.

Table 3.1: Channel configuration properties

Property	Values	Description
tag	string	Indicates the channel TAG.
type	BFT or CFT	Indicates que QoP of the channel.
totalOrder	0 = inactive; 1 = active	If active, all events inside the channel are totally ordered across all replicas. If inactive, events inside this channel may have different orders from replica to replica.
maxPublishers	integer value >= 0	Maximum number of publishers for the channel.
maxSubscribers	integer value >= 0	Maximum number of subscribers for the channel.
maxEventsPer Queue	integer value >= 0	Maximum number of events that each output queue may have.
eventsDischarging Order	OLDER or NEWER	Discarding order of events in case an output queue becomes full.

Channels are managed according to a control table. Each one has a unique table that contains: the identification of the authorized publishers and subscribers, the size of the output queues, the order in which the events (if necessary) should be discarded, the QoP and QoS of the channel and its topic also called TAG.

In Figure 3.5 we have an abstract representation of a channel with two publishers and three subscribers. Events sent by publishers are placed in the input queue of the channel. Then, they are filtered, processed and routed by routing algorithm to the output queues accordingly to the control table. Since multiple subscriptions by the same subscriber to the same channel are not allowed, there is one output queue per subscriber.

If subscribers don't process events at the rate that they are being created, in order to minimize the loss of events, channels offer an event buffer. This buffer takes the form of limited size output queues. The size of the queues can vary from channel to channel. Once a queue is full, events start to be discarded according to the selected method.

The FIT-Broker offers two different fault tolerance protocols. The first protocol deals with crash faults. Publishers will publish their events in all the replicas, but subscribers will receive the event from the fastest replica. If no ordering is required the service will be available if at least one replica is executing. If ordering is required the service will be available if at least $f+1$ replicas are up and running.

The second protocol deals with Byzantine faults. By using this protocol, clients will publish their events in all correct replicas. In order to be able to tolerate Byzantine faults in the broker service, the clients use a voting mechanism that allows them to distinguish a correct event from an incorrect one if a majority of replicas are correct.

An abstraction of the voting mechanism is represented in Figure 3.6. If no ordering is required, the service will be available if at least $f+1$ broker replicas are available. In cases where total order is required, the service will be available if at least $2f+1$ replicas are available.

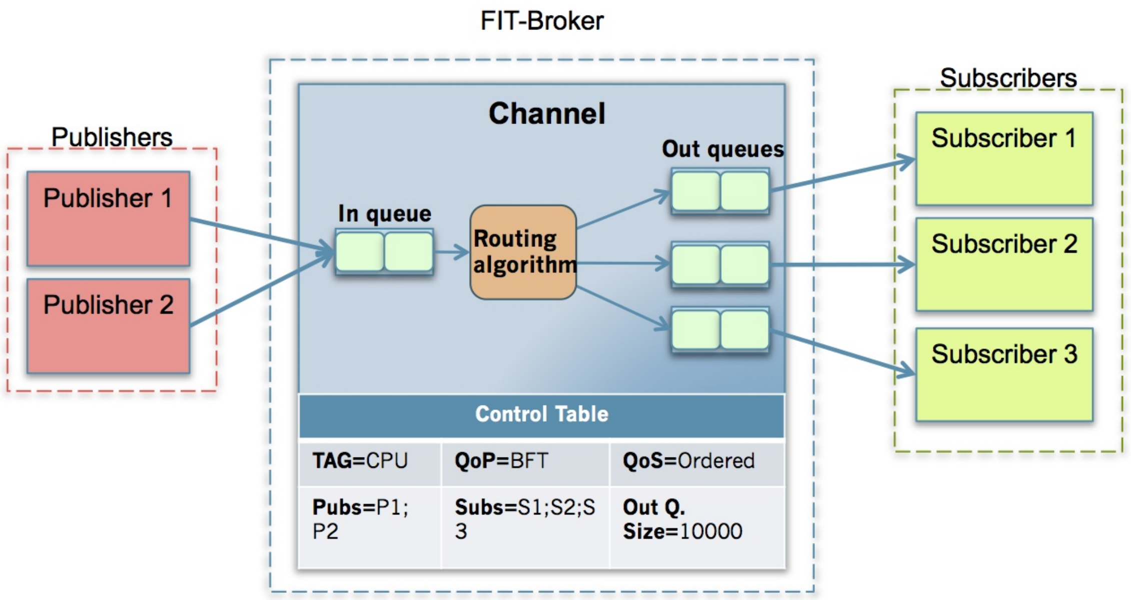

Figure 3.5: FIT-Broker channel

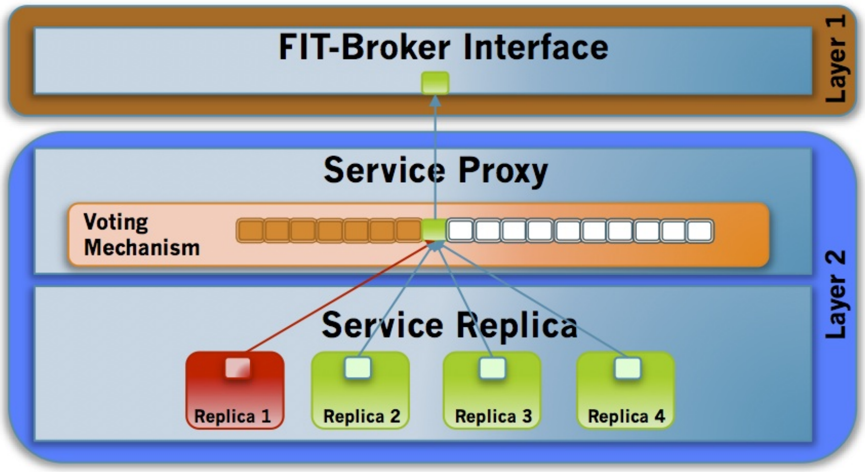

Figure 3.6: Voter mechanism abstraction

To provide total order mechanisms to both fault tolerance protocols, we have decided to use BFT-SMaRt [4]. BFT-SMaRt is a state of the art framework that implements a leader driven protocol that is based on the SMR concept and is documented on [33].

3.3.3 Data structures

An event is the most basic data unit of the FIT-Broker. It contains probes monitoring data and five extra control fields as represented in Figure 3.7. The first one is the event ID, that is unique and identifies the event. The next field is the client ID which is also unique and identifies the creator of the event. The timestamp marks the moments in which the event was created and can be used for control purposes like validity of the monitored information. The next field identifies the event in the channel, meaning that it indicates the position of the event inside the output queues. The last control field indicates if it is an urgent event, indicating if it has priority over the other events or not thus satisfying a QoS requirement.

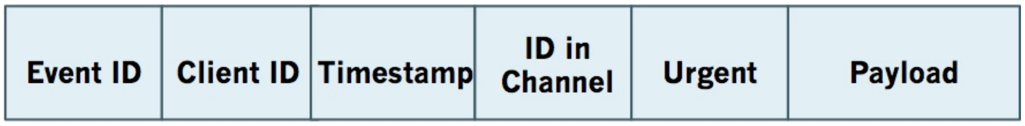

Figure 3.7: Event structure

Events are transported in requests that contain the information required to invoke operations like publish or subscriber on the FIT-Broker. They are composed of several fields as represented in Figure 3.8. Client ID is unique and identifies the client that created the request. Channel TAG contains the TAG that the request is concerned (recall that the TAG is like a topic, as used in publish-subscribe systems). The operation fields contains the requested operation. Replica ID identifies the replica where the request was generated. Operation status states if the requested operation was successful. The sequence number is a unique number that identifies the request and its corresponding response. Number of events indicates the amount of events that the request contains. The last field contains the events that the request transports.

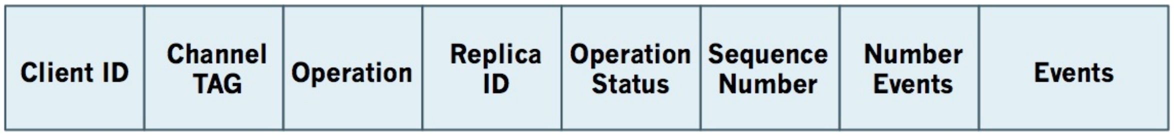

Figure 3.8: Request structure

In fact, it is possible to send several events in a unique request message, which may be very relevant for performance and scalability reasons. It is also possible to send zero events, such as when operations like register and unregister need to be performed.

Chapter 4

FIT-Broker: Design and Implementation

This chapter is dedicated to the implementation of the FIT-Broker. The first section describes the software components that compose the FIT-Broker and how they interact with each other. Then we describe BFT-SMaRt, which is a fundamental building block in our work, as it implements the necessary protocols for BFT replication. Other implementation options are presented after that, namely the option of using Netty as a communication library, and Java as the programming language to build the FIT-Broker components. Finally, we describe how we implemented the necessary protocols, namely for communication between the clients and the FIT-Broker replicas, for ordering client events, for leader election among replicas, among others.

4.1 Software components

In the following section we describe the FIT-Broker software components in a layer by layer approach. As explained in section 3.3.2, the FIT-Broker is composed of layers. Each layer itself can contain of one or several software components.

4.1.1 Layer 1 - The FIT-Broker interface

Layer 1 is composed by the FIT-Broker interface. It allows client interaction with the FIT-Broker by offering different methods to the client. To interact with the FIT-Broker, a client just has to create an object using the class **MessageBrokerClient.java** and invoke the methods.

Clients invoke methods from the interface, which in turn creates a request that is passed to the next layer. Once a method is invoked, the clients block until a response is returned.

When a response is ready to be delivered, the interface receives it from layer 2 and delivers it to the client.

A complete list of the methods that are offered by the interface can be found in Appendix D.

4.1.2 Layer 2 - Service proxy and Service replica

Layer 2 is responsible for the communication between clients and replicas. On the client side we have the service proxy that is composed by two components and sends requests over the network to service replicas. Each FIT-Broker replica contains a service replica that is responsible for receiving the clients requests and is composed of up to four different threads, one for each possible combination of QoP and QoS. Figure 4.1 represents the interaction inside layer 2.

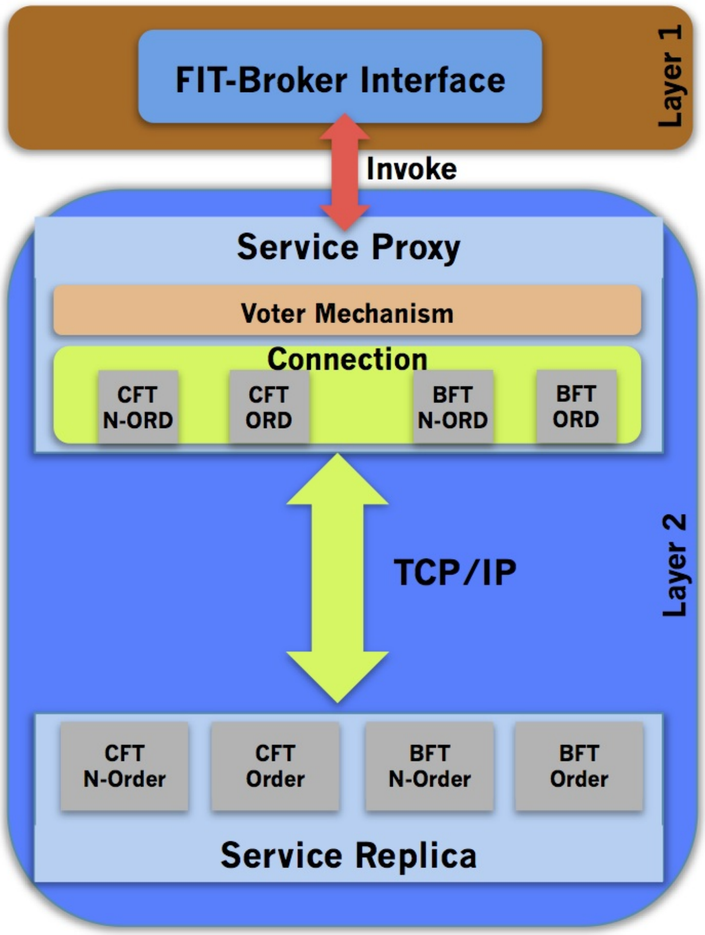

Figure 4.1: Service proxy and Service replica connection

Service proxy

The service proxy is composed by two elements: *a)* a voting mechanism; *b)* a connection that represents the connections to each replica. The voting mechanism implements a voting algorithm and is used when necessary to verify the responses from replicas. Each service proxy can have one and only one type connection from the possible four that are available(represented in figure 4.1). The type of connection is specified using configuration files.

Additionally, to be able to enhance the FIT-Broker in the future, the service proxy offers an interface called `ClientSideConnection`. This interface allows the addiction of new types of connections. The methods specified by the interface are presented in Appendix E.

In order to be able to receive and deliver request to layer 1, the service proxy offers a set of methods. The available methods are detailed in Appendix F.

When a client first initiates, it reads the configuration files, and uses that information to build the service proxy. Table 4.1 contains the configuration, their possible values and the corresponding description.

The additional configuration files are: *a)* System.config that specifies the number of servers and faults; *b)* hosts.config that contains a pair IP/port for each replica that composes the FIT-Broker. Table 4.2 contains the parameters of the file System.config and listing 4.1 is an example of the hosts.config.

Listing 4.1: Example of hosts.config file

```
#––– ip port –––
#Wed Oct 31 17:01:10 WET 2012
127.0.0.1 1010
127.0.0.1 1020
127.0.0.1 1030
127.0.0.1 1040
```

Service replica

Each replica contains a service replica that is responsible for five things: *1)* receive the request from clients; *b)* send responses to the clients; *c)* if necessary, order the requests using ordering protocols; *d)* perform leader election when required; *e)* execute the state transfer protocol if necessary.

To accomplish its job, the service replica contains four different server sockets, one for each possible combination of QoS and QoP. Upon initiation, a replica reads the configuration files and creates a thread for each type of socket specified in the configuration

Table 4.1: Client configuration parameters

Property	Values	Description
maxNumberOf EventsToFetch PerRequest	integer value >= 0	Maximum amount of events that can be retrieved for each request. If set to 0 (unlimited), retrieves all events from the output queues of replicas.
numberOfEvents ToCachePer Request	integer value >= 0	Specifies the maximum number of events cached per request.
useOrdered	0 = inactive; 1 = active	If inactive, an unordered connection will be used. If set active, it will use an ordered connection.
useCFT	0 = inactive; 1 = active	If active, a CFT connection will be used between the service proxy and service replicas.
useSBFT	0 = inactive; 1 = active	If active, a BFT connection will be used between the service proxy and service replicas.
cftConfigPath	string = path	Specifies the folder that contains the files with the IP address and port of each replica and a configuration file specifying the number of faults and replicas for the CFT connection.
BftConfigPath	string = path	Specifies the folder that contains the files with the IP address and port of each replica and a configuration file specifying the number of faults and replicas for the BFT connection.

Table 4.2: System.config parameters

Property	Values	Description
system.servers.num	integer value >= 0	Number of FIT-Broker replicas.
system.servers.f	integer value >= 0	Max number of faults.
system.initial.view	id,id...	The identification of the FIT-Broker replicas.

files. The configuration file parameters are presented in table 4.3. The table presents the configuration parameters, the possible values that they can assume and their description.

Each server socket runs in a separated thread meaning that they run concurrently. Each time a new client connects, a thread dedicated to that client is created. This allows that multiple clients use the same type of connections and perform concurrent operations in the FIT-Broker.

When a request is received, if it was received through the socket that is used when requests require total order, the service replica will execute the corresponding protocol. Upon the completion of the protocol, the request is delivered to layer 3. Once delivered, the thread waits until a response is ready to be sent back to the client.

When leader election or state transfers are required, the replicas stop responding to clients (new request are stored and current ones are put on hold) and perform the required operations. Once completed, the replicas resume normal operation (resume responding to clients and process the stored requests).

Table 4.3: Server configurations parameters

Property	Values	Description
channelConfigPath	string = path	The path to the folder containing the channel files.
BFTorder	0 = inactive; 1 = active	Indicates if the server connection that receives QoS = total order and QoP = BFT should be used.
BFTnoOrder	0 = inactive; 1 = active	Indicates if the server connection that receives QoS = unordered and QoP = BFT should be used.
BFTconfigPath	string = path	Specifies the folder that contains the files with the IP ad- dress and port of each replica that are executing this server connection and a configuration file specifying the maximum number of faults and the number of replicas.
CFTorder	0 = inactive; 1 = active	Indicates if the server connection that receives QoS = total order and QoP = CFT should be used.
CFTnoOrder	0 = inactive; 1 = active	Indicates if the server connection that receives QoS = unordered and QoP = CFT should be used.
CFTconfigPath	string = path	Specifies the folder that contains the files with the IP ad- dress and port of each replica that are executing this server connection and a configuration file specifying the maximum number of faults and the number of replicas.

As the service proxy, the service replica also offers a way to add new server sockets. This is done by implementing the interface `ServerConnection`. This interface extends the `Runnable` interface indicating that the new server connection should be a thread and must implement the method `run()`. The interface contains two methods, the `close()` method that closes the server socket releasing all associated resources and the `run()` method that should start the thread processing.

4.1.3 Layer 3 - Event broker service and Data service

Layer 3 is responsible for event routing. In this layer, events created by publishers are received and delivered to the corresponding channels. Inside the channels, the events are then routed to the output queues of subscribers. Figure 4.2 shows a representation of layer 3.

Event broker service

The Event broker service receives requests sent by clients and invokes the corresponding operation like publish or unregister over the data service.

Once an operation is invoke, the Event broker service waits for the corresponding result. Upon the reception of the result, it generates a new request with the same sequence number, adds the result to it, and sends it to the client through layer two. The response request is identified with the same sequence number in order for the clients to be able to match the response with the sent request.

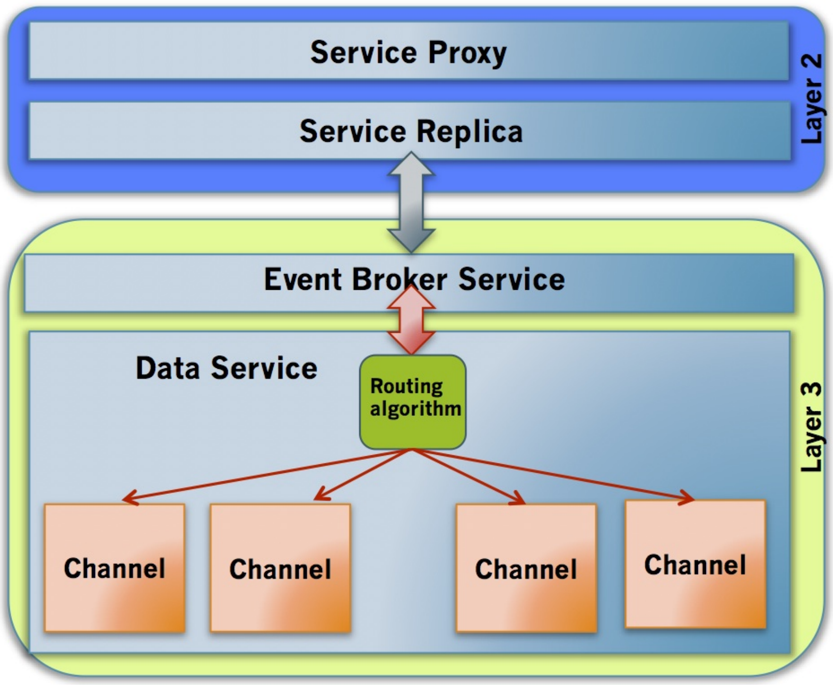

Figure 4.2: Data service

Data service

The data service is composed of channels. It can contain one or more channels. Figure 4.2 is a representation of the data service. Channels are created when a replica is initiated using static configuration files that are equal in all replicas. Table 3.1 contains the detailed parameters of the channel configuration files.

Inside the data service, events are routed to their channels. Inside the channels, events are copied to the output queues of subscribers.

To be able to interact with it, the data service offers a set of methods like `insertNewEvent(E, TAG)` or `insertNewSubscriber (SUB, TAG);` to the event broker service. The complete list of methods is described in Appendix G, and are defined by the interface `Storage`. They allow for new and different data services to be added to the FIT-Broker. This is useful if users want to use different types of channels.

By using the methods in listing G.1, it is possible to see that the FIT-Broker routes the events based on topics. In our framework, the topic is called TAG. In order to allow for other types of routing inside the channels, we offer an interface called `Channel` that allows for the creation of other types of channels. Appendix H shows the offered interface that contains methods like `addSubscriber (SUB)` and `addEventToSubscriber(E, SUB)`. With it, users can for example create channels where the routing of events is done by analysing the content of the event and not just the TAG.

By using the `Storage` and the `Channel` interfaces, it is possible to create new data services that can respond to the different needs of each user.

4.2 BFT-SMaRt

BFT-SMaRt is a high-performance BFT SMR library developed in Java [33]. It provides the tools necessary to build client-server applications assuming a Byzantine fault model. BFT-SMaRt provides client-server communication channels using Netty and replica-replica communication using simple socket communication. An application built using this framework must be composed of at least *3f+1* replicas to be able to tolerate up to *f* faults. Replicas must be built using the SMR paradigm described in section 2.5.1, meaning that replicas must perform deterministic operations.

BFT-SMaRt uses the VP-Consensus protocol [33] as a grey-box (Figure 4.3). This allows a modular implementation of SMR without requiring a reliable broadcast to ensure that all replicas have the same requests, and thus avoiding the extra communication steps that normally would be required.

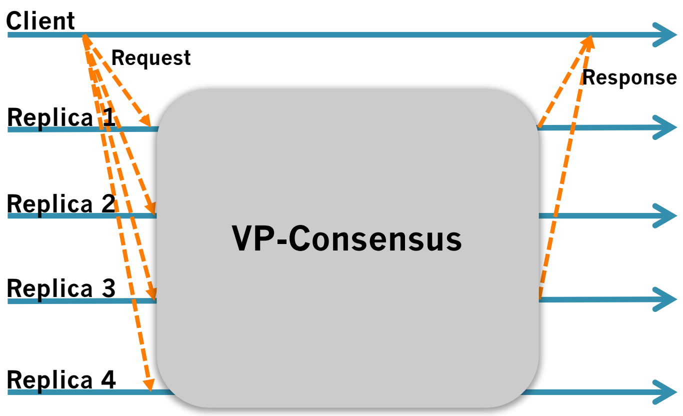

Figure 4.3: VP-Consensus as grey-box

This framework follows a modular architecture called MOD-SMaRt presented in figure 4.4. It is responsible for assuring SMR and it utilizes the VP-Consensus to execute an agreement on the requests to be delivered at each step to the application. A reliable and authenticated communication layer is used to guarantee point-to-point message delivery. The communication layer is of general purpose, meaning that it is used for VP-Consensus and communication between processes inside the system.

The FIT-Broker makes use of BFT-SMaRt as a support framework that provides ordering, state transfer and leader election protocols when the chosen QoS is total order. But since

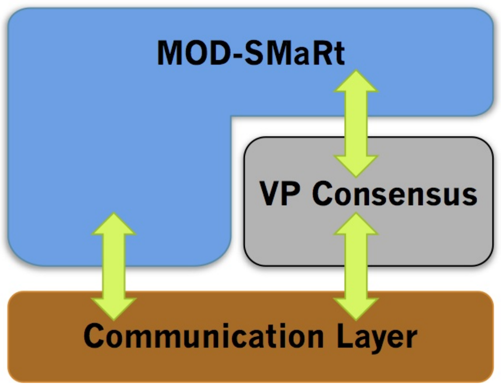

Figure 4.4: MOD-SMaRt

BFT-SMaRt is only intended to be used when assuming a Byzantine fault model, it became necessary to make modifications to the framework and adapt its protocols, in order to also be used when assuming a Crash fault model.

4.3 Other implementation options

In this section we address two important decision options, namely the option to use Netty as a communication library, and Java as the programming language.

Java

Java is an object oriented programming language that was specifically designed to have as few implementation dependencies as possible [26]. It allows programmers to follow the write once run anywhere (WORA) model. This means that a program developed on one platform can be executed in a different platform without making any changes to the code or recompiling the program. This portability is due to the fact that a Java program is compiled to Java bytecode (class file) which is intended to be interpreted using a Java Virtual Machine (JVM). JVMs are currently available for most of the platforms, thus a program that was built using Java can run in any of those platforms.

We have chosen Java as our programming language due to the fact that BFT-SMaRt was implemented using it. This makes the job of integrating with BFT-SMaRt less complicated that it would be if another programming language was used. Also, by using Java we also gain the advantage of portability.

Java's portability presents itself as a natural way to solve the diversity issue. By using Java, the FIT-Broker replicas can run in different platforms without requiring alterations to the code. In doing this, we are taking advantage of the fact that different platforms have different vulnerabilities, and thus possibly making the job of an attacker much more harder. On the recovery of a replica, it is possible to execute it in a different platform even if on the same physical hardware. In this way, if the attacker was using a vulnerability in the platform to compromise the replica, possibly, it was removed. Also, different replicas may run different versions of the JVM or run the JVM with different arguments, thus generating different stacks and making an attacker jobs that much more difficult.

To develop the FIT-Broker, we have used Java 1.7.X.

Netty

Netty is a Java framework that provides an asynchronous and event-driven API intended to build highly concurrent client-server applications [28]. It is build over Java New Input/Output (NIO) API and it makes use of high performance threads pools to receive, send, and process information. By using and managing the thread pool, the Netty framework allows us to build applications that are efficient regarding resource consumption.

Netty makes use of channels to provide communication between clients and servers. A channel is composed of several objects: *a)* a channel pipeline composed of several handlers each with is own specific function; *b)* an object encoder; *c)* object decoder; **d)** a sink used as a buffer to send and receive messages. Additionally, each channel has two streams, the downstream and the upstream.

Handlers in Netty are used to perform operations inside a channel. Developers may create as many as they wish and add them to the channel pipeline. For example, it is possible to have one that encodes the content of a message, while another generates a cryptographic hash to assure integrity of the message. Messages are passed from one handler to another, and once the message is passed, the previous is free to receive another message. The use of handlers is a very useful, it allows a clear separations between operations and by adding or removing them from the channel pipeline it is possible to add or remove new operations without requiring major alterations to the programs code.

The object encoder and object decoder are used to convert objects to bytes and and bytes to objects respectively to be sent and received. Their use is not mandatory, however it is encouraged by Netty. According to Netty creators, the use of an object encoder and decoder that knows how to convert bytes to objects and objects to bytes, greatly improves the performance of the program.

The sink is used to send and receive messages. Once a message is ready to be sent, it is passed in a non-blocking way to the sink that is responsible for writing it in the socket. If the socket is busy, it stores the message until the socket it free. When receiving messages, the sink calls the object decoder and passes to it the received bytes.

The streams inside channels describe the path that a message has to travel inside a channel, in other words, it specifies the handlers and the order in which they are used to send or receive a message. The downstream is used to send messages, while the upstream is used to receive messages. An important factor that should be mentioned is that both streams may share handlers, meaning that the same handler can be used to send and receive messages.

An abstraction of Netty channels is presented in Figure 4.5. Both client and server channels are practically identical. They can be composed of the same type of components. The main different between them is that, a client channel is unique, meaning that one client can only have one channel, while a server will use the channel factory to create a new channel each time a new client connects to him.

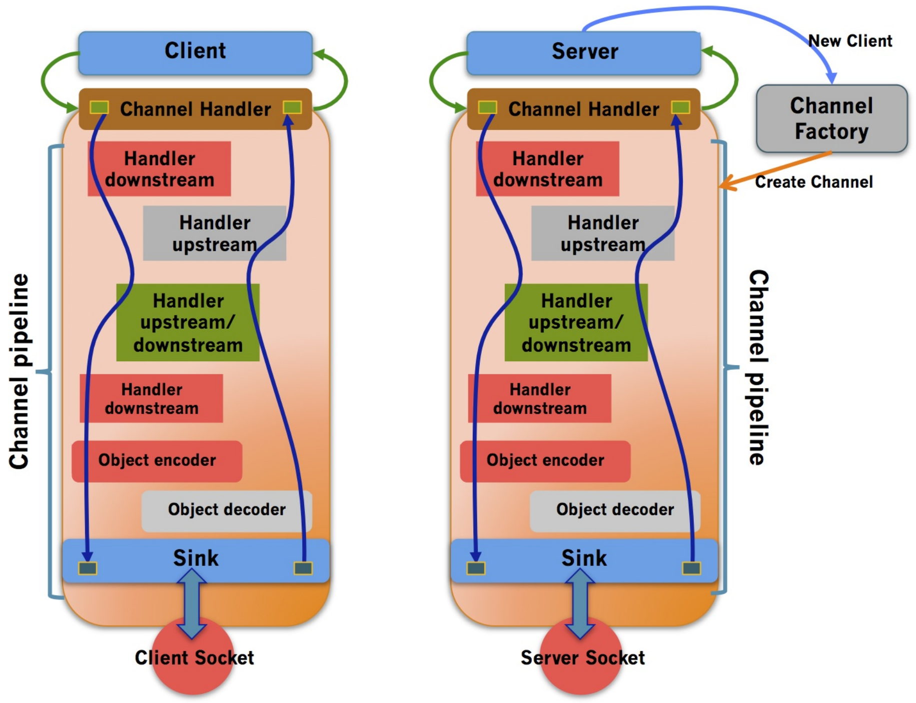

Figure 4.5: Netty Channels

The FIT-Broker makes use of Netty due to the fact that BFT-SMaRt uses it to provide client-server communication. But, since BFT-SMaRt is only used when the QoS is total order, we decided to also utilize Netty as our framework to provide reliable client-replica communication when the QoS is unordered. To do so it was necessary to implement new communication channels following the Netty channel model.

We have chosen to use version 3.6.5 of Netty to implement reliable client-replica communication.

4.4 Protocols and algorithms

Each channel in the FIT-Broker has an associated QoS and QoP. To satisfy these requirements, different protocols are required. All protocols are implemented in layer 2, as explained in the previous section.

The communication protocols are used by the clients to send messages to the FIT-Broker and receive the corresponding reply. There are two different communication protocols, one for CFT and another for BFT.

To order events, the FIT-Broker uses two different protocols. One protocol is used when the QoP is CFT, while another is used when the QoP is BFT. These protocols are executed between the replicas and are implemented inside the BFT-SMaRt framework.

To be able to tolerate and recover from faults, satisfying the QoS and QoP of channels is not enough. If replicas fail and are recovered it is necessary to transfer the correct state to these replicas. In order to do it, we require state transfer protocols.

Since the ordering protocols that are provided by BFT-SMaRt are leader driven, it can happen that failed replicas were leaders, and as such it becomes necessary to elect a new leader. To perform this election, we require leader election protocols, which are also provided by BFT-SMaRt.

For the FIT-Broker, we have developed the unordered communication protocols, modified the BFT ordering protocols provided by BFT-SMaRt to adapt them to the Crash fault model, and used the protocols provided by BFT-SMaRt. For the ones that we have developed, they are explained and accompanied by their algorithms. The ones that we have modified are briefly described and their corresponding algorithm can be found in the appendix sections. The ones that are used directly from the BFT-SMaRt library, we make a brief description and refer the reader to the original article for the corresponding algorithm.

This section is structured as follows: first we describe the communication protocols, next the ordering protocols, followed by the state transfer protocols and for last the leader election protocols.

4.4.1 Communication protocols

In this section, we describe the communication protocols that are executed between clients and replicas. First we describe the crash fault tolerant communication protocol that is used when events only need to be resilient against crash faults. Next we describe the Byzantine fault tolerant communication protocol that is used when it is assumed that Byzantine faults like intrusions, and events must be propagated despite those faults.

Crash fault tolerant communication protocol

This protocol is designed to provide service in the presence of crash faults. With it, replicas can fail by stopping and may be recovered any number of times provided that at least one replica is correct.

If using this protocol, a client sends its requests to all the replicas. All replicas reply with a response message. The client delivers the first response it receives and discard all the remaining redundant responses from the other replicas. Figure 4.6 represents this protocol. This protocol requires a minimum of *f+1* replicas to tolerate up to *f* faults.

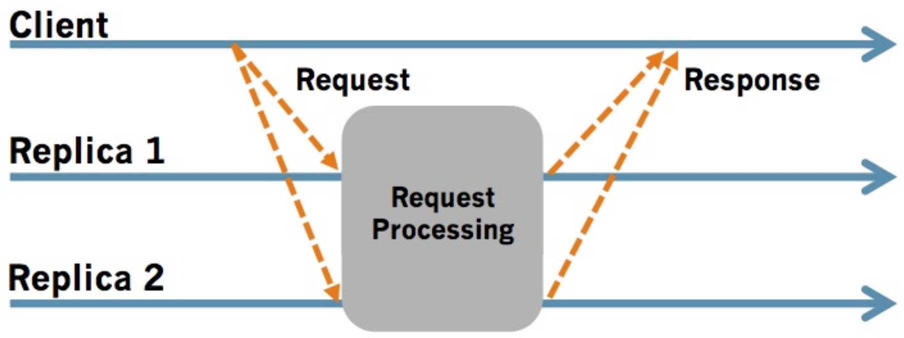

Figure 4.6: Crash fault tolerant communication protocol

Algorithm 1 describes the implementation of the protocol. It is composed by four methods:

Initialization: this method is executed by the clients when a new service proxy is created. It creates a *listOfPendingRequests* that will contain the *id* of the requests that whre sent by the client and are waiting a response, and a list that contains the identification of the replicas that contain CFT channels.

Notification: this method is invoked for two reasons: *a)* new replica is available; *b)* a replica has crashed and is no longer available. Both reasons will insert or remove a replica from the list of replicas.

Invocation: the client interface calls the method **invoke()** of the service proxy. It sends the request to each replica contained in the list of available replicas. It adds the *sequence number* of the request to the list of pending requests. The client blocks until a response is returned.

Response reception: once a response is received, it is delivered to the client. All other redundant responses are discarded.

Algorithm 1: CFT communication (executed on the service proxy)

```
//invoked by the client upon start of new service proxy
Upon init do
    $listOfPendingRequests \leftarrow \emptyset$
    $listOfReplicasCft =$ buildListOfReplicasCftFromConfiguration()

//invoked by the client upon the reception of a replica notification
Upon reception of replica notification do
    updateListOfReplicasCft()

//invoked by the client to send a request
Upon invoke(Request) do
    foreach $r \in listOfReplicasCft$ do
        send($Request$) to $r$
    addToListOfPendingRequests($Request.seqNumber$)

//invoked by the client upon the reception of a request
Upon reception of (RESPONSE, $RequestResponse$) from $r \in listOfReplicasCft$ do
    if isInListOfPendingRequests($ResquestResponse.seqNumber$) then
        RemoveFromListOfPendingRequests($ResquestResponse.seqNumber$)
        DeliverToClient($RequestResponse$)
    else
        Discard($ResquestResponse$)
```

Byzantine fault tolerant communication protocol

The Byzantine fault tolerant communication protocol is design to provide service in the presence of arbitrary faults. With it, replicas may suffer any type of faults.

A replica is considered correct if it does not deviate from protocol and performs the operations according to their specifications. Also, it is assumed that replicas may fail and be recovered an infinite number of times.

Figure 4.7 exemplifies the protocol. The protocol is very similar to the crash fault tolerant communication protocol. The differences are in the number of required replicas that increases to *2f+1*, the minimum number of correct replicas that are necessary is *f+1*, and now the client waits for *f+1* equal responses before considering that the response is correct.

Algorithm 2 describes the implementation of this protocol and is composed by the same four methods that the crash fault tolerant communication protocol was. The difference is in the method *reception of* that makes use of a BFT svoting algorithm to be able to know if it can deliver the response.

The BFT voter algorithm is described in algorithm 3 and is used to verify if a quorum of equal responses to a request have been received. For simplicity, the vote method was divided into two different methods, the *vote* that votes on requests to verify if the have

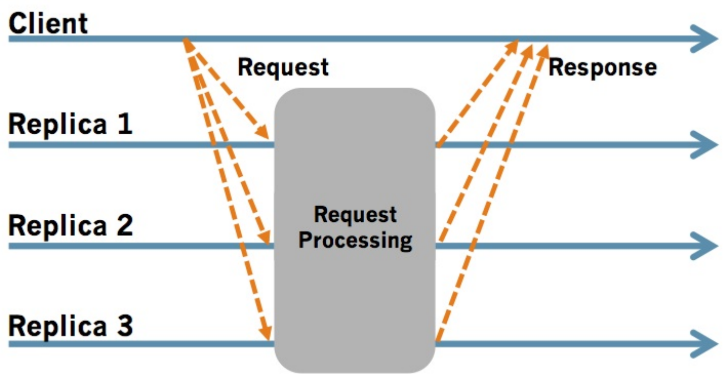

Figure 4.7: Byzantine fault tolerant communication protocol

reached a quorum and the *voteOnEvents* that verifies if events have reached a quorum. As such, the algorithm is composed by four methods:

Initialization: executed when a new *bftVoter* is created. It creates a *listOfReceivedRequests* to hold requests until the minimum number of equal requests needed for the quorum is reached, a *listOfReceivedEvents* containing the events that haven't reached quorum and a variable to store the *id* of the responses;

Awaited response: sets the *id* of the response that is currently awaited;

Vote: this method is invoked when a response is received. It verifies if a minimum of *f+1* equal responses have been received, if so returns a response and removes it from the *listOfReceivedRequests*, otherwise returns void;

Vote on events: this method is invoke when a response is ready to be delivered. It returns all events that have reached a quorum of *f+1* equal events and deletes them from the *listOfReceivedEvents*.

4.4.2 Ordering protocols

To provide order, we have chosen to use the *Validated and Provable Consensus* (VP-Consensus) [33]. It is a leader driven protocol that allows a modular implementation of SMR without using reliable broadcast, thus avoiding the extra communication steps required to ensure that all requests have reached all replicas. The VP-Consensus protocol is implemented on the BFT-SMaRt framework.

An illustration of the VP-Consensus is presented in Figure 4.8. The leader is responsible for providing order to the requests. Each interaction of the protocol is identified with an execution id (EID) and is composed by three phases:

Algorithm 2: BFT communication (executed on the service proxy)

```
//invoked by the client upon start of new service proxy
Upon init do
    $bftVoter = $ initiateBftVoter()
    $listOfReplicasBft = $ buildListOfReplicasBftFromConfiguration()

//invoked by the client upon the reception of a replica notification
Upon reception of replica notification do
    updateListOfReplicasBft()

//invoked by the client to send a request
Upon invoke(Request) do
    foreach $r \in listOfReplicasBft$ do
        send($Request$) to $r$
    bftVoter.awaitedResp($Request.seqNumber$)

//invoked by the client upon the reception of a request
Upon reception of (RESPONSE, $RequestResponse$) from $r \in listOfReplicasBft$ do
    $votedResponse = bftVoter$.vote($RequestResponse$)
    if $votedResponse \neq \emptyset$ then
        DeliverToClient($votedResponse$)
```

Algorithm 3: BFT voter (executed in the service proxy)

```
// invoked by the BFT algorithm when it starts Upon init do
    $listOfReceivedRequests \leftarrow \emptyset$
    $listOfReceivedEvents \leftarrow \emptyset$
    $awaitedReply \leftarrow \emptyset$

// invoked by the BFT algorithm before sendin a request Upon awaitedResp($seqNumber$) do
    $awaitedReply \leftarrow seqNumber$

// invoked by the BFT algorithm upon the reception of a request Upon vote($Request$) do
    $listOfReceivedEvents$.add($Request.events$)
    $Request$.removeAllEvents()
    if $awaitedReply$.equals($Request.seqNumber$) and reachQuorum($Request$) then
        $listOfReceivedRequests \leftarrow$ new empty list
        $awaitedReply \leftarrow \emptyset$
        $Request$.addListOfEvents(voteOnEvents())
        return $Request$
    else
        $listOfReceivedRequests \leftarrow$ add $Request$
        return $\emptyset$

// invoked by the Voter algorithm when a request has reached a quorum Upon voteOnEvents do
    $finaListOfEvents \leftarrow$ new empty list
    foreach $e \in listOfReceivedEvents$ do
        if reachEventQuorum($e$) then
            $finaListOfEvents$.add($e$)
            $listOfReceivedEvents$.remove($e$)
    return $finaListOfEvents$
```

Propose: the leader proposes an order to the requests by broadcasting a PROPOSE message to all replicas;

Weak: each replica confirms the reception of the PROPOSE message by broadcasting a WEAK message. Once a replica receives $\frac{n+f}{2}$ WEAK messages it broadcasts a STRONG message;

Strong: once a replica receives $f + 1$ STRONG messages, it can accept the proposed order because it is certain that at least $f + 1$ correct replicas have accepted the propose message.

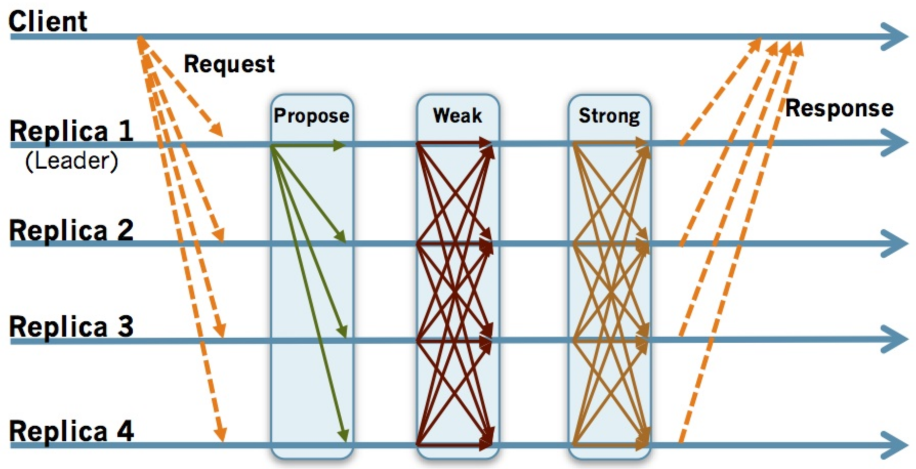

Figure 4.8: VP-Consensus

VP-Consensus is intended to be used when Byzantine faults are considered. Since the FIT-Broker can be used for channels requiring only crash fault tolerance, some modifications to the protocol were made for the case were CFT channels are used and order is also required. Thus the two versions have been implemented. Figure 4.9 illustrates the modified protocol. When used for CFT, the VP-Consensus is composed by two phases:

Propose: the leader proposes a order to the requests by broadcasting a PROPOSE message to all replicas. Each replica on receiving a PROPOSE message broadcasts a STRONG message;

Strong: once a replica receives $f + 1$ STRONG messages it accepts the proposed order.

The algorithm that represents the implementation of the VP-Consensus for CFT is presented in Appendix A.

The VP-Consensus is implemented in BFT-SMaRt [4], as such, we have decided to use it as our support framework to provide total order. BFT-SMaRt is described later in section 4.2.

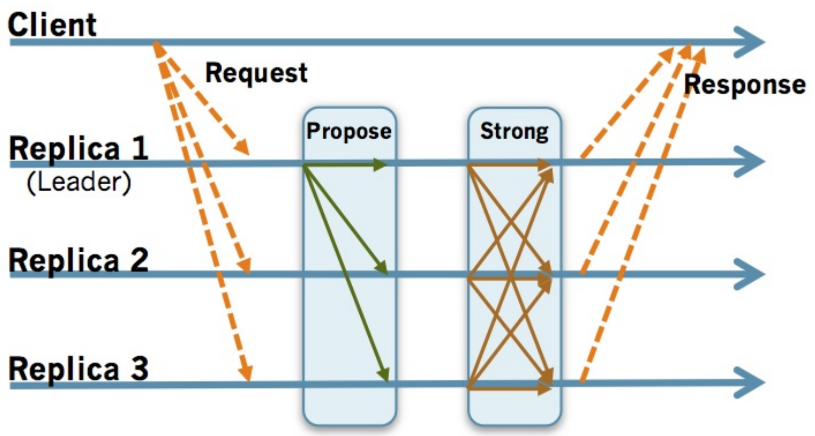

Figure 4.9: VP-Consensus modified for CFT

4.4.3 Leader election

The ordering protocols are leader driven, and since leadres may crash, we require protocols to include ways for electing a new leader in case the current one fails. BFT-SMaRt provides a leader election protocol to be use in conjunction with VP-Consensus [4].

Figure 4.10 represents the message pattern of the leader election protocol. It is divided into two different phases: the leader election phase and the synchronization phase. In the leader election phase, a new leader is elected by all replicas using a deterministic operation. In the synchronization phase, all replicas are synchronized to the current state of the system since some of them may be in a previous execution.

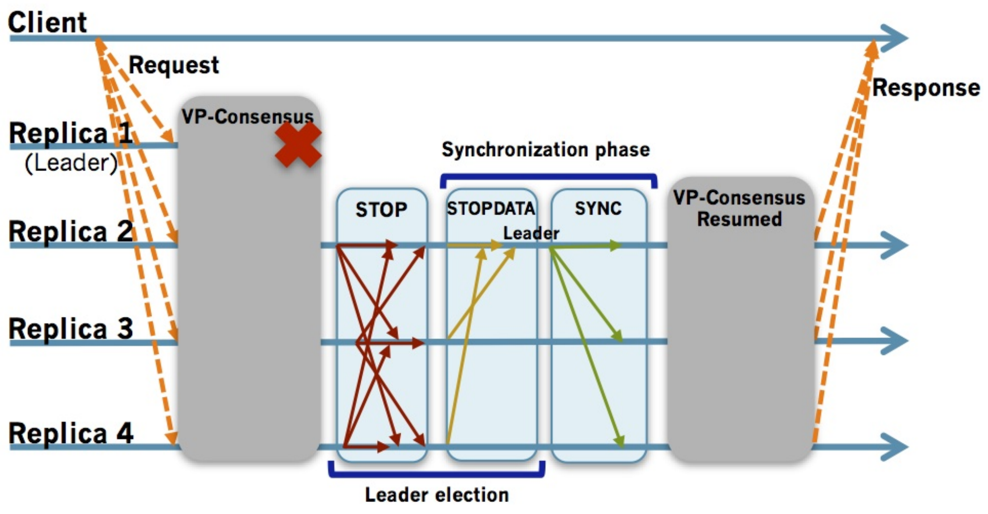

Figure 4.10: Leader election

Since we have a modified version of the VP-Consensus to work in CFT, we also require a new leader election protocol adapted to the CFT case. The protocol is initiated when the *timeout* for a request in a replica is triggered, meaning that the leader has not proposed an order for that request. The replica will broadcast a STOP message. Once a replica receives a STOP message, it will perform a deterministic operation electing the new leader. Figure 4.11 represents the protocol.

In Appendix B we provide the algorithm that represents the implementation of the leader election protocol for CFT.

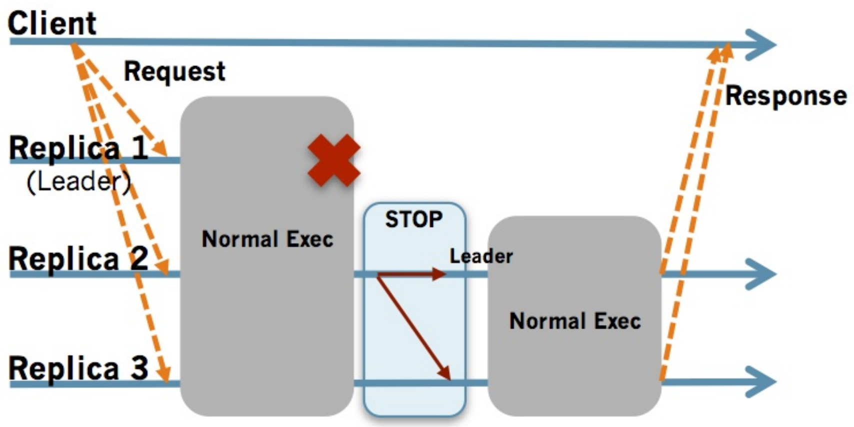

Figure 4.11: Leader election

4.4.4 State transfer

State transfers are required for two reasons: *1)* a replica is late and needs to be updated with the current state of faster replicas; *2)* new or recovered replica has joined the system and needs to know the current state in order to be able to provide service. The state transfer protocol allows a correct state to be transferred to a replica, thus updating the replica to the current system state.

The FIT-Broker has two different states. The first state contains the channels that don't offer total order as a QoS, and is composed of: *a)* the current existing channels; *b)* registered clients.

The second state contains the channels that offer total order. This state is composed of all existing total order channels, the registered clients, and all the events that are currently inside the channels.

A different state transfer protocol is required for each type state. Depending on the channels that are being offered by the FIT-Broker, it may be required to execute both of them or only one. For example if the FIT-Broker contains only total order channels, only the ordered state transfer protocol is executed, but if it contains ordered and unordered channels, both protocols are executed.

Unordered state transfer

The unordered state is composed by two separate things: *a)* registered clients; *b)* existing channels.

The protocol is divided into two distinct steps. The first step is concerned with the channels. Since channels are created from static configuration files, when a replica recovers, it reads the information in the configuration files and creates them. This is performed before allowing clients to connect. Client connections are only allowed after all channels are created. This operation is presented in algorithm 4. The algorithm contains one method that reads the configuration files and creates the channels.

Algorithm 4: Replica initiation

```
Upon init do
    $listOfChannels$ = buildListOfChannelsFromConfiguration()
    acceptClientConnections()
```

The second is the client registration. If a replica fails and recovers, the client proxy will detect the recovery and resend the registration request to it. Algorithm 5 presents the implementation of this step, it is composed by three methods:

Initialization: this method is executed when the client is initiated. It creates a list containing all available replicas and a variable to record the crucial operations;

Send: this method is invoked by the client to broadcast a request to all replicas. If the request is a register or unregister operation, then it is recorded. The method finally broadcasts the request to all replicas;

Update list of replicas: this method is executed when a replica fails or recovers. It updates the list of replicas and, if necessary, re-sends the register operation.

In Figure 4.12 we have a representation of the two steps of the state transfer protocol. Replica three fails and recovers. The client detects the recovery (if a replica fails, the client is always trying to reconnect to it on the same IP and port) and re-sends the register operation.

Ordered state transfer

With respect to ordered state transfer, we use the protocol already implemented and provided by BFT-SMaRt. There are two distinct states: the BFT-SMaRt state and the FIT-Broker state.

Algorithm 5: Register algorithm

```
Upon init do
    $listOfReplicas = \texttt{buildListOfReplicasFromConfiguration()}$
    $recordedOperation \leftarrow \emptyset$
Upon send ⟨Request, op⟩ do
    if $op = register$ then
        $recordedOperation \leftarrow op$
    if $op = unregister$ then
        $recordedOperation \leftarrow \emptyset$
    foreach $r \in listOfReplicas$ do
        send($Request$, r)
Upon updateListOfReplicas ⟨Replica, r, status⟩ do
    if $status = online$ then
        send($recordedOperation$, r)
        $listOfReplicas \leftarrow listOfReplicas \bigcup$ ⟨Replica, r, status⟩
    else
        $listOfReplicas \leftarrow listOfReplicas$ / ⟨Replica, r, status⟩
```

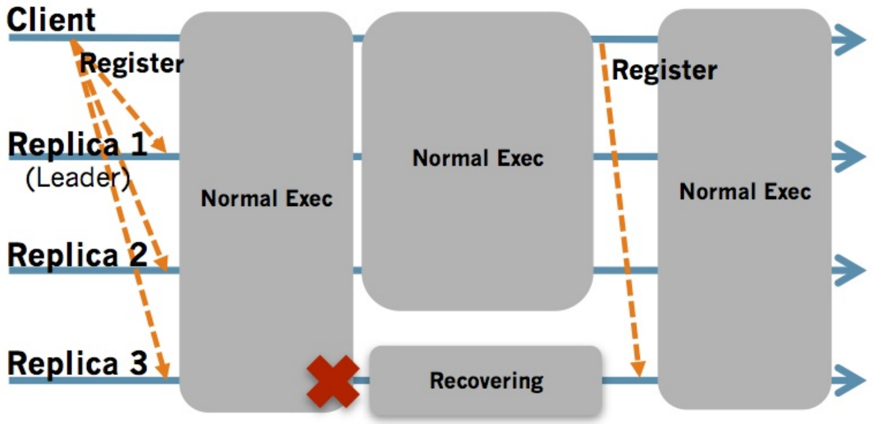

Figure 4.12: State transfer

The state transfer protocol will transfer both the sate of the FIT-Broker (i.e, channels, configuration, etc) and also the state of the ordering protocol implemented within BFT-SMaRt. More specifically, the state of the BFT-SMaRt ordering protocol includes: *a)* current execution ID; *b)* last checkpoint; *c)* execution log. The FIT-Broker state is composed of: *a)* last checkpoint (content of the Data Service in layer 3 when checkpoint was asked); *b)* execution log since last checkpoint.

In this section we first describe the state transferring protocol for CFT followed by the protocol for BFT, both of them used for transferring state information relative to channels providing total order.

Crash fault tolerance

This state transfer protocol is an adaptation of the protocol presented in [33] that is implemented by BFT-SMaRt.

Figure 4.13 shows the message pattern of the protocol. Replica 2 detects that it is outdated (receives the messages of the ordering protocol from other replicas and detects that it is late) and request a state transfer from the leader and starts to store the received requests from the clients. Upon the reception of the request, the leader replies with the state. Replica 2 updates the state, processes the stored requests and continues the current execution. The algorithm in Appendix C represent the implementation of the protocol.

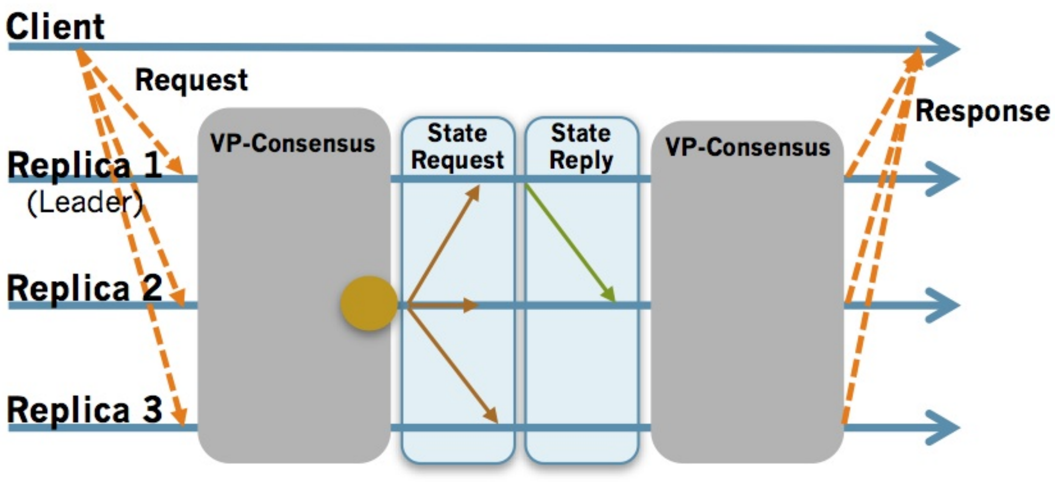

Figure 4.13: CFT state transfer

Byzantine fault tolerance

In order to recover from faults, the BFT-SMaRt implements a state transfer protocol described in [33].

Figure 4.14 represents the message pattern of the protocol. In the case where a replica detects that it is outdated regarding other replicas, it stops all its current executions and broadcasts a message requesting the current system state. Upon receiving the request, the replicas will reply with the current state. However, since states can be very big and their transfer can take a long time, only the leader transfers the complete state, all other replicas generate a digest (hash) of their state and transfer it. Upon the reception of *f+1* replies, the replica generates a hash of the state that it has receivedand compares it to the other hashes. If they match, it then can use the state to update itself and resumes execution.

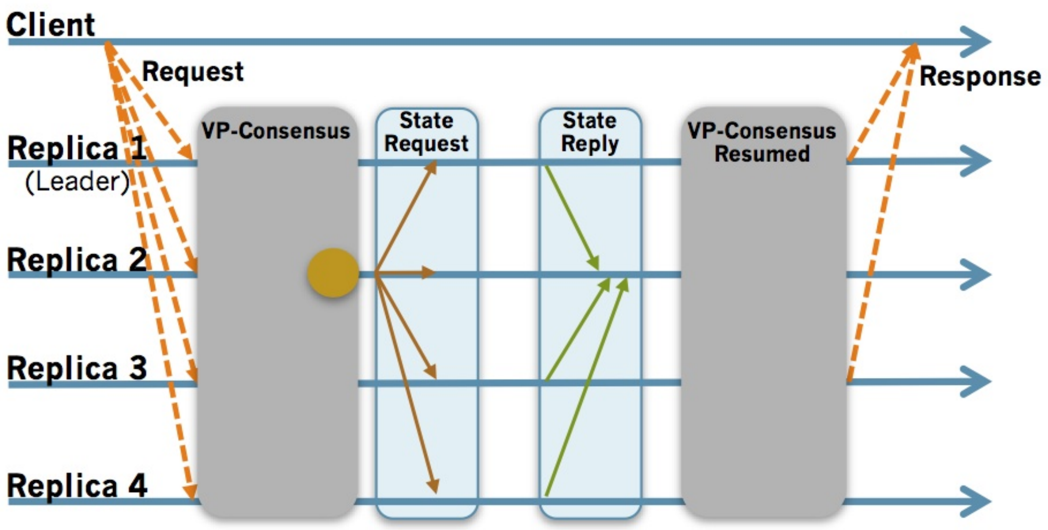

Figure 4.14: BFT state transfer

Chapter 5

Experimental results and evaluation

This chapter describes the performance evaluation of the FIT-Broker in a local-area-network (LAN).

We start by describing the test environment, followed by the description and results of the performed tests. Next we present a proof of concept application, and lastly we have a small discussion about the tests.

5.1 Testbeds

Our test environment was composed by six machines connected by a local network. One of the machines contained an older hardware while five had more recent hardware.

The older hardware was a Dell PowerEdge 850 with an Intel Pentium 4 CPU at a clock speed of 2.8GHz, 2GB of memory, two Broadcom NetXtreme BCM5721 to connect to the LAN, and was running Ubuntu 10.04. It was used to run the subscribers (described in section 5.2). We chose to use an older hardware firstly because of schedule issues with other users of the infrastructure, and second because subscribers tend to be less active then publishers. Meaning that subscribers tend to access the message brokers less times then publishers but consume more information each time they do.

The second type of machines where Dell PowerEdge R410. Each of them with two Intel Xeon E5520 processors at 2.27GHz of clock speed, 32GB of memory, two Broadcom NetXtreme II BCM5716 to connect to the LAN, and Ubuntu 10.04 as their operating system.

All machines where interconnected through a Gigabit switch.

5.2 Performed tests and results

We have performed several throughput tests to the FIT-Broker. These tests had the objective of evaluating the behaviour of the FIT-Broker in several different situations. We started by testing the throughput during normal execution, followed by a continuous crash and recovery test. Next we injected a fault making a replica lie about the outcome of an operation. After that we increased the number of replicas and thus increased the number of tolerated faults, finally, we have tested sending events in batch.

The throughput test was executed for every possible combination of QoP and QoS. For each other test, and since we consider that the worst performing case is when the required QoS includes ordering and the QoP is BFT, we have set the QoP to BFT and vary the QoS to compare the difference.

Each event used in tests had the size of 253 bytes, and its payload was composed of 9 random bytes generated using the local clock and the id of the publisher. A request with one event had the size of 716 bytes.

All tests were performed using ten channels. Due to the fact that monitoring systems tend to have less monitoring consoles and more probes and agents, each channel had two subscribers and up to ten publishers.

Since we execute less subscribers then publishers, we have chosen to use the oldest hardware to run them. Each subscriber consumes all events from its output queue four times per second. This settings where chosen due to the fact that although subscribers consume more information then publishers, they tend to be less active, meaning that subscribers aren't always consuming information.

Each test starts with one publisher per channel and is scaled to ten publishers per channel. As such, we have executed them in one of the machines with the powerful hardware. In order to increase the number of publishers, we add a new publisher to each channel. This means that at any given time the total number of publishers can be calculated by multiplying the number of publishers per channel by the number of channels. Each test starts with one publisher per channel and is scaled up to ten publishers per channel. Each publisher tries to publish events as fas as possible.

The remaining four machines were used to execute the FIT-Broker replicas. Each one contained one replica.

For all tests except the batch test, we generated twenty million events that are evenly distributed by the publishers. For the batch test, since events are cached, we have increased the number of events to one billion due to the fact that the previous number was not enough to get a clear reading of the throughput. Table 5.1 shows the number of events per publisher for all tests except for the batch test.

For the test results, only the publishers throughput is considered. This is due to the fact that each successful publish operation means that events were routed to the corresponding output queues and are waiting to be consumed. All tests are conducted as follows: publishers are created and generate the events, then they start to publish events at the same time, and the total time it took to publish all events is measured. Once events are published, the throughput is calculated by dividing the total number of events by the time it took to publish them.

Table 5.1: Number of events per publisher

Number of publishers per channel	**Number of events per publisher**
1	2000000
2	1000000
3	666666
4	500000
5	400000
6	333333
7	285714
8	250000
9	222222
10	200000

5.2.1 Throughput test during normal execution

The throughput test during normal execution had the objective of discovering the maximum number of events that can be routed per second for each combination of QoP and QoS.

To perform this test, we executed each combination of QoP and QoS separately. For all combinations we have set $f = 1$. Table 5.2 details the number of replicas used for each combination. Each replica was executing normally, and no faults were injected.

Table 5.2: Number of replicas when $f = 1$

QoP and QoS	**Number of replicas**
QoP = BFT; QoS = Ordered	4
QoP = BFT; QoS = Unordered	3
QoP = CFT; QoS = Ordered	3
QoP = CFT; QoS = Unordered	2

Figure 5.1 contains the graphic representation of the results. It is possible to see that the highest throughput is when the QoS is unordered and the worst is when QoS is ordered. This is due to the fact that ordering protocols require that replicas communicate with each other which requires extra time and resource consumption. The difference in the throughput between CFT and BFT for both QoS comes from the fact that for CFT fewer replicas are required and it only waits for one answer before proceeding. Whilst, for BFT more replicas are required, and each client has to wait for $f + 1$ equal responses before proceeding. Another important fact is that we have the highest throughput between two and five publishers per channel. This may be due to the fact that we have reached the maximum capacity of the machines to execute concurrent operations. Table 5.3 contains the detailed values that we have obtained.

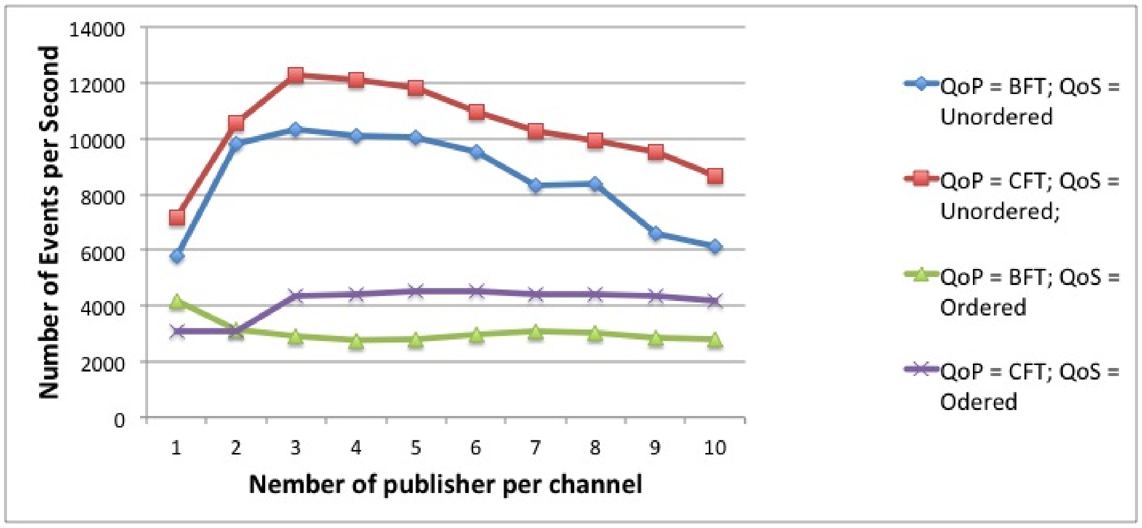

Figure 5.1: Throughput test

Table 5.3: Throughput test results

Number of Publishers per channel	QoP = BFT; QoS = Unordered	QoP = CFT; QoS = Unordered	QoP = BFT; QoS = Ordered	QoP = CFT; QoS = Ordered
1	5797	7194	4201	3081
2	9803	10582	3120	3114
3	10362	12269	2936	4357
4	10101	12121	2770	4424
5	10052	11834	2808	4535
6	9523	10989	2962	4555
7	8298	10256	3095	4439
8	8403	9950	3007	4415
9	6622	9523	2867	4364
10	6116	8695	2797	4185

5.2.2 Throughput test during replica crash and recovery

For this test we periodically crashed a replica and rebooted it causing a state transfer to the new replica. Our objective was to compare the performance degradation between a normal execution and when state transfer protocols are being executed.This test was performed for both QoS and the QoP is BFT.

To perform this test we had one of the replicas running for one minute, crashed it and rebooted it again, thus triggering a state transfer to the new replica. The fact that the replica runs for a minute gives enough time to complete the state transfer protocol and induces extra traffic in the network. We performed this action during the entire test and always in the same replica. Figure 5.2 depicts the results of the test and the accurate values can be found in Table 5.4.

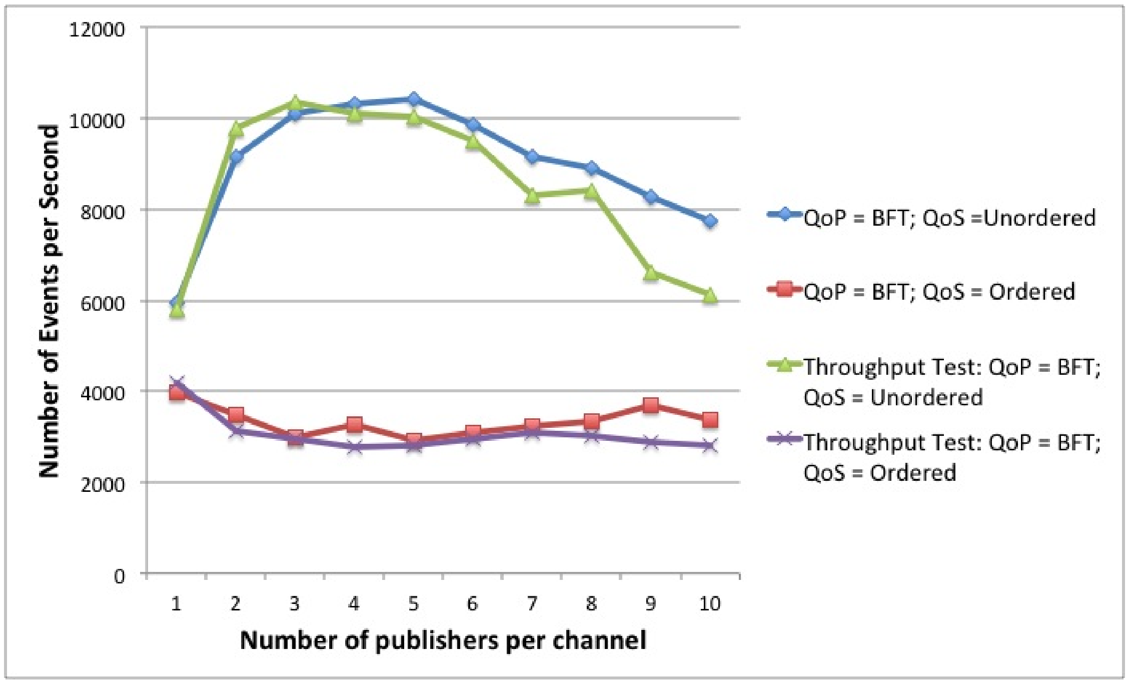

Figure 5.2: State transfer protocol throughput test

By comparing the values with the Throughput Test values, we can see that there is few or no degradation in the service. In fact, in some cases we can observe some performance enhancements. This can be explained by the fact that when the QoS is unordered, the state transfer protocol is a simple re-register message. When the QoS is ordered, and since we have subscribers consuming their output queues, the state tends to be small since it contains few events. Another factor is that, when a replica is down, publishers have to send one less message to publish an event. Also, there are fewer replicas participating in the ordering protocol thus reducing the number of messages in the network. Note, however, that although the results are sometimes better, the system is running without any redundancy redundancy and is thus unable to tolerate further faults.

Table 5.4: State transfer throughput test results

Number of Publishers per channel	QoP = BFT; QoS = Unordered	QoP = BFT; QoS = Ordered
1	5934	3960
2	9174	3472
3	10101	2994
4	10309	3257
5	10416	2923
6	9852	3110
7	9174	3255
8	8928	3338
9	8264	3676
10	7751	3387

5.2.3 Replica lying throughput test

For this test we injected a Byzantine fault in one of the replicas. This replica does not perform the requested operations, refuses to participate in ordering protocols by sending incorrect information to the other replicas, and sends incorrect responses to the client proxy.

In Figure 5.3 we illustrate the outcome of the tests. As can be seen, there is little performance degradation in comparison with the tests performed in 5.2.1. This is due to the fact that there are still enough replicas participating to ensure that clients receive a correct service. Also, there is no performance enhancement because we still have all replicas responding to the requests, meaning that we have the same quantity of messages in the network. Table 5.5 details the precise results of the test.

Table 5.5: Replica lying throughput test results

Number of Publishers per channel	QoP = BFT; QoS = Unordered	QoP = BFT; QoS = Ordered
1	5996	4032
2	9442	3016
3	10218	2812
4	10173	3039
5	10025	2822
6	9467	3189
7	8679	2985
8	8502	2657
9	7104	2594
10	6763	2566

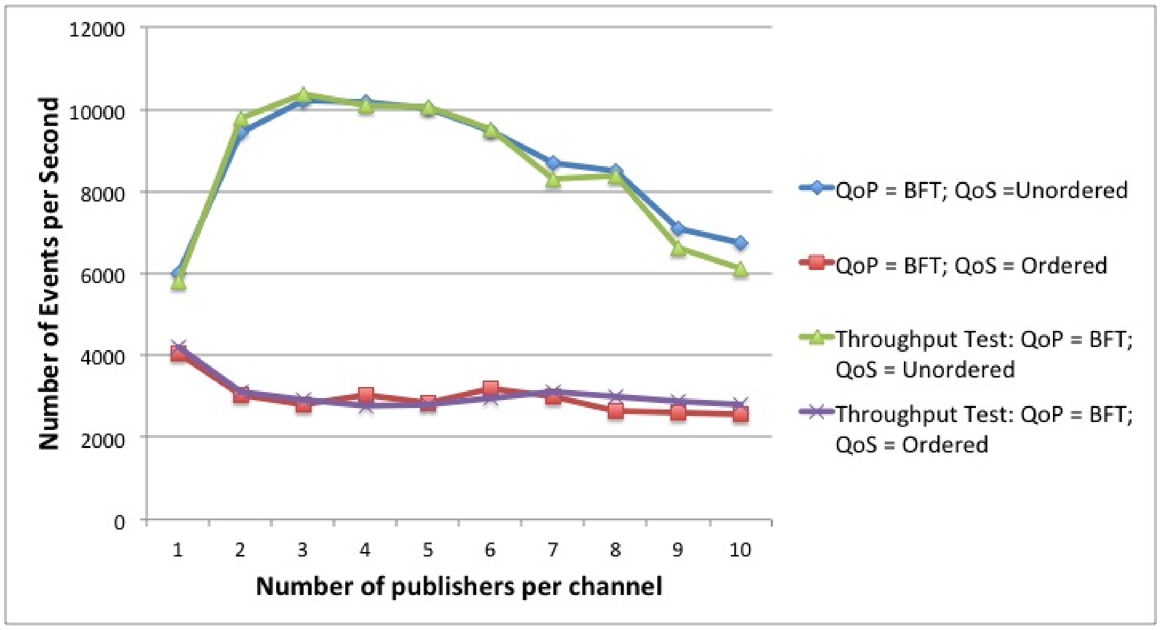

Figure 5.3: Replica lying throughput test

5.2.4 Higher resilience number

For this test, we have set $f = 2$, which means that we have a higher resilience number. This test requires seven replicas $(3f + 1)$ when the requested QoS includes ordering and five replicas $(2f + 1)$ for unordered. All replicas were executing without any fault or failure during the entire test.

We can see in Figure 5.4 that increasing the resilience number greatly affects the throughput of the system. This is due to the fact that we have more replicas, and thus more messages in the network.

When the QoS is unordered, the client has to send more messages to publish an event. When the QoS is ordered, there is not much degradation in the service due to the fact that executing the ordering protocol is heavy and takes time, thus having more replicas will not affect the overall performance. Also, a client has to wait for more responses from different replicas before considering that they have a correct response.

As the number of clients of the FIT-Broker grows, the results show that when the QoS is unordered, the throughput decreases to values similar to the ones of the ordered QoS. This can be explained by the fact that we are running all clients in one physical machine and using a single network interface thus creating a big concurrency to access it. We are confident that by separating the clients, we will see the throughput increase. The detailed results can be found in Table 5.6.

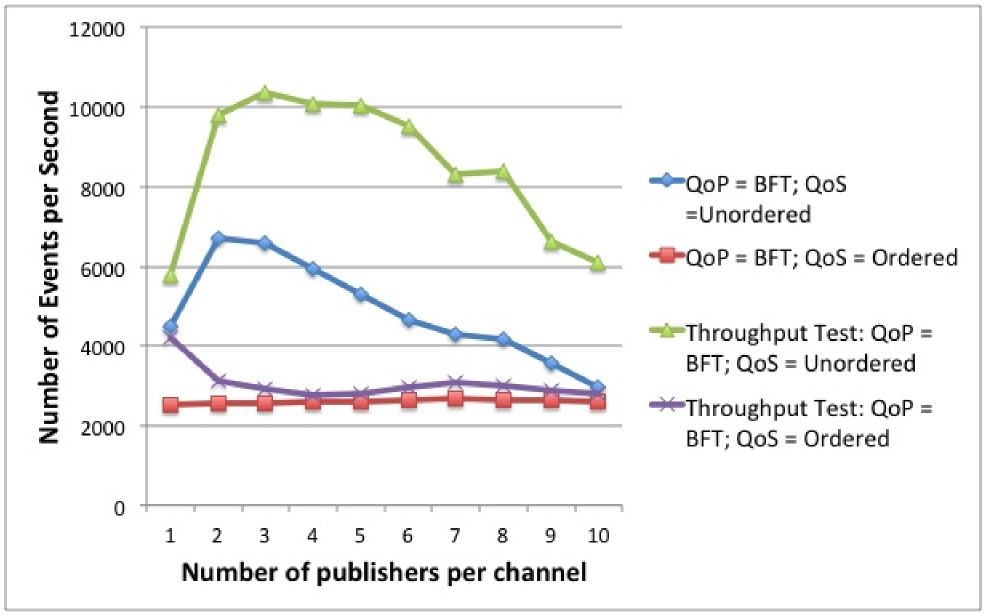

Figure 5.4: Higher resilience throughput test

Table 5.6: Higher resilience throughput test results

Number of Publishers per channel	QoP = BFT; QoS = Unordered	QoP = BFT; QoS = Ordered
1	4474	2525
2	6688	2547
3	6578	2577
4	5934	2604
5	5291	2612
6	4662	2624
7	4291	2667
8	4175	2645
9	3577	2636
10	2954	2603

5.2.5 Batch test

The batch test measured the throughput of the FIT-Broker when caching of events is used. This can be an useful feature for clients that can or want to publish or retrieve several events in each communication.

To perform this test, we increased the number of generated events to one billion events. For the duration of the test, we cached one hundred events and sent them in a single request.

Each request had 7329 bytes. The size of the request is smaller than one hundred events (100x253bytes) because although the events are different, they may share some information. For this reasons, instead of replicating that information, Java creates different pointers (which are small) to the same information.

Figure 5.5 shows that by caching events, we reach much higher throughputs in terms of events that are routed inside channels. These results are obtained because each request carries multiple events, thus reducing the number of messages required to publish multiple events. The detailed results of the test is presented in Table 5.7.

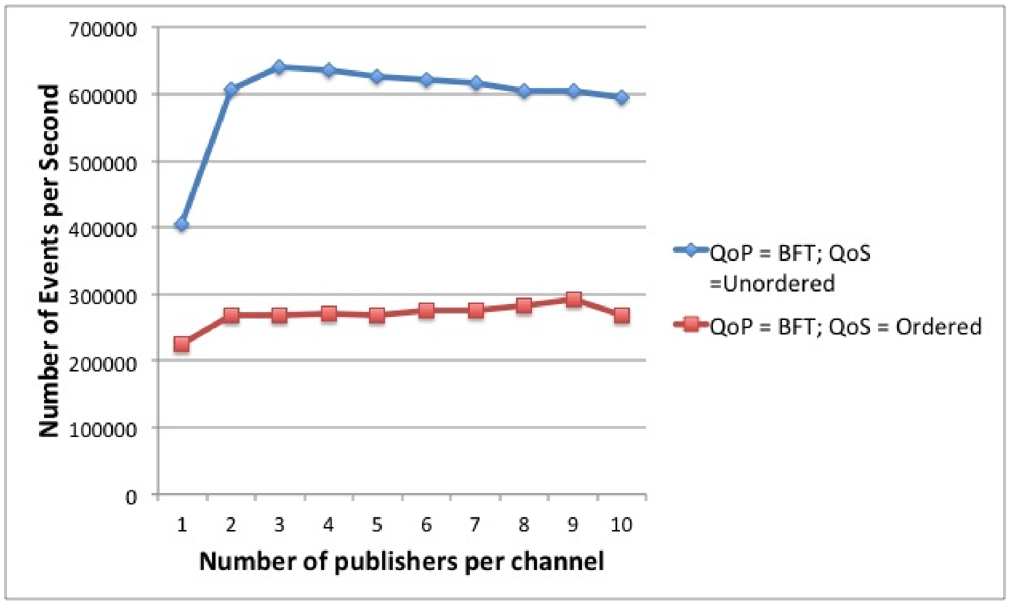

Figure 5.5: Batch test throughput

Table 5.7: Batch test throughput

Number of Publishers per channel	QoP = BFT; QoS = Unordered	QoP = BFT; QoS = Ordered
1	405515	225225
2	608272	267379
3	641436	268817
4	636537	271739
5	626174	268817
6	620732	274725
7	618046	276243
8	604229	282485
9	603864	292397
10	595238	267737

5.3 Proof of concept application

This section presents a small application that was developed as a proof of concept. It was used to provided simple demonstrations of the FIT-Broker functions and reliability.

The application was composed of two things: *a)* a client interface that creates clients and shows the throughput *b)* a simple interface capable controlling and injecting faults in the replicas.

Client interface

The client interface is composed of two parts: *a)* a logical part that creates and runs clients of the FIT-Broker; *b)* a graphical part that show the current the current information of the clients.

When initiated, the interface creates a set of clients that connect with the FIT-Broker. The number of clients and the channels TAGs are as an argument when the interface is initiated. The clients connected to the channels using a round robin algorithm. For example, if there are four channels and four clients, each channel will have one client, but if we have four channels and six clients, two of the channels will have two clients and the other ones will have only one.

There are two graphical interface. Both are presented in Figure 5.6. The only thing that distinguishes them is the fact that one is used for subscribers and is green and the other is has a light red color and is used for publishers. They are composed by the same two elements. The first is a speedometer that shows the current throughput. The second element is responsible for showing the number of events published/consumed so far.

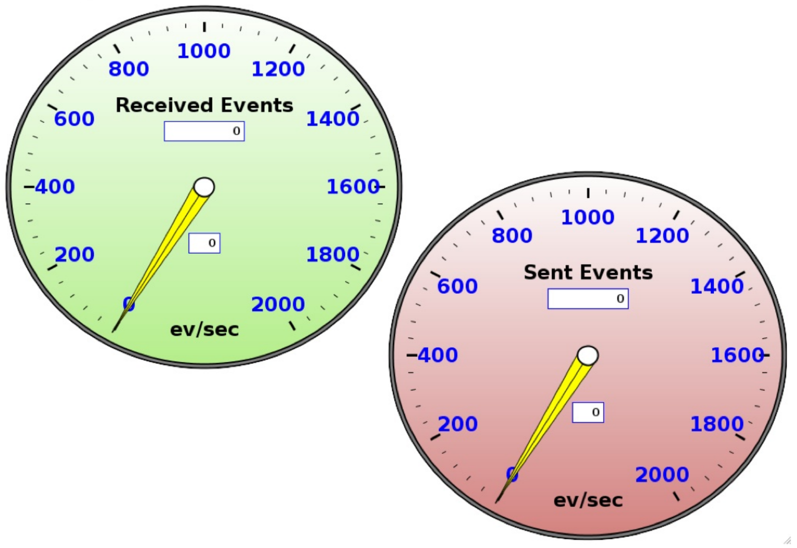

Figure 5.6: Clients interface

Replica controller

The controller has three different functions: *1)* start replicas; *2)* stop replicas; *3)* inject faults. The injected faults can have two different natures: *a)* slow down a replica thus simulating a denial of service (DoS) attack; *b)* simulate a Byzantine behaviour by making the replica participate correctly in the ordering protocols but sending random bits as responses to the clients.

The controller interacts with each replica using a TCP/IP connection through which it sends commands to a receiver thread present in each replica. In Figure 5.7 we have a representation of this interaction. In this case, the controller connects to all replicas.

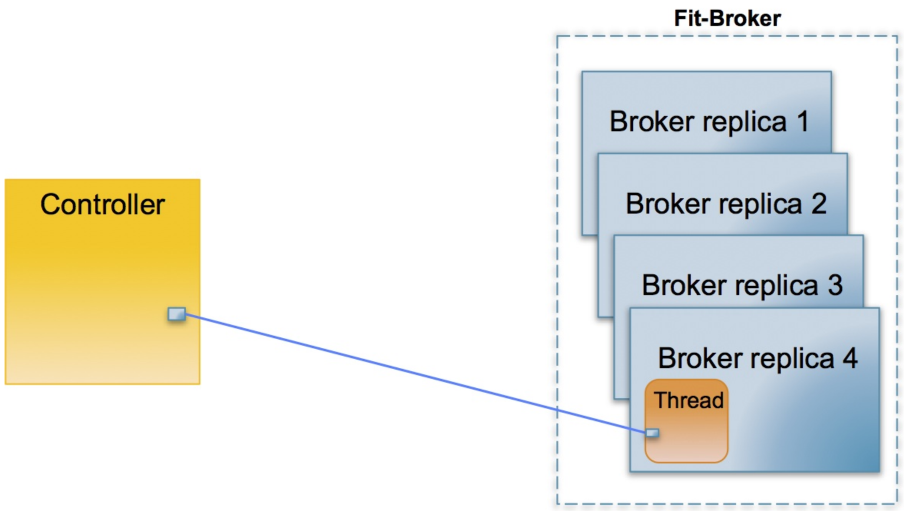

Figure 5.7: Controller connection over TCP/IP

The controller is presented in Figure 5.8. For simplicity, the controller is divided into rows. Each row commands the replica with the id that is indicated in the left side. This first button is the start button, and it initiates the replica. The second button is the stop button, it stops the replica. The third button makes the replica have a Byzantine behaviour. The last button slows down the replica trying to simulate a DoS attack. The coloured dot that is positioned on the right side of each row indicates the current status of the replica and can assume four colors: *a)* green if the replica is executing normally; *b)* red if the replica is stopped: *c)* yellow if the replica is slow; *d)* black if the replica is performing a Byzantine behaviour.

The start and stop button can be used to trigger protocols like leader election or state transfer. For example, if we stop the current leader, we will trigger a leader election protocol execution. If we restart the replica, it will detect that it is late regarding the others and will trigger a state transfer protocol. Both these examples are reflected in the client interface through a throughput decrease.

Each fault that is inject has a different effect on the FIT-Broker. The Byzantine behaviour as stated, makes the replica send random bits as responses to the clients. The recovery of this case has to be triggered manually since we don't make any assumptions about the behaviour of clients.

By slowing down the replica we are simulating cases where the replica is suddenly bombarded with requests from clients. This can happen for two reasons: *a)* there is a peek of clients and all replicas receive the same requests and consequently, the system slows down; *b)* an attacker has flooded the replica with requests thus creating a DoS attack to it. In the second case, when a replica detects that it is in a previous execution regarding the other replicas, it triggers a state transfer to "jump" to the current system state. This can alleviate the effects of the attack because all the request that where in memory before the state transfer are discarded.

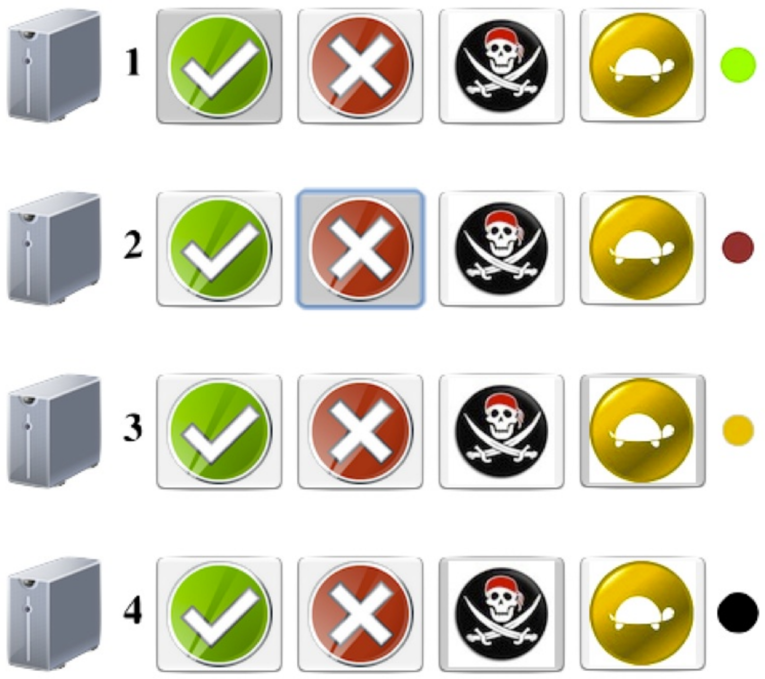

Figure 5.8: Controller interface

5.4 Discussion

In this chapter we have presented a list of tests and their corresponding results. By looking at them, we can conclude several things.

The first conclusion is that ordering protocols have a heavy toll on the throughput. In all tests it is possible to see a big difference between both QoS's. This last affirmation is not true in the case of higher resilience when the number of clients grows. As stated before, we are confident that running the clients in different machines will increase the throughput.

From uniting the facts: *a)* ordering protocols require extra communication; *b)* caching greatly increases the throughout; we can conclude that communication represents a big part of the time it takes to interact with the FIT-Broker. In the case of ordering protocols, we see it in every graphic, meaning that the difference between the QoS's is the extra communication of the ordering protocol. In the second case, we are sending requests that are about ten times greater than when sending requests with a single request.

Another conclusion can be reached by dividing the results in table 5.7 by one hundred, is that for ordered QoS the values don't vary that much. However for unordered QoS, the values are smaller than on normal operation. This means that both the size of the request and the number of events per request have an impact on throughput.

Another important conclusion is that we reach the maximum throughput values for QoS unordered between two and five clients per channel. This may mean that we have reached the maximum capacity of parallelization of the machines used to run the tests. Maybe running the tests in machines with more parallelization power will reveal higher values. On the other hand we can conclude that for ordered QoS the throughput always stabilizes around three thousand events peer second. Once again this is due to the fact that ordering requires extra communication, thus leaving the channels with long periods of inactivity while the order is decided.

Form the test presented in 5.2.2 and 5.2.3 we can conclude that the presence of faults has little impact on the throughput. This is due to the fact that there are still enough correct replicas to assure a correct service.

Additionally, in this chapter we have presented a small application that is used as a proof of concept. It demonstrates that the FIT-Broker is capable of performing correctly in the presence of faults. Also, by using it, we can demonstrate the execution of all the implemented protocols.

Chapter 6

Conclusion

6.1 Final remarks

This thesis has presented the implementation and evaluation of a reliable event broker middleware that is capable of providing multiple levels of quality of service and protection. It provides simple client interfaces so that existing monitoring systems (or other systems) can integrate with it.

We have created a layered architecture for the FIT-Broker that clearly separates the components by their function. Each component offers a set of methods that are specified by interfaces. This allows clients to modify or create new components to enhanced or adapt our framework to better serve the needs of their systems.

We have explain our implementation decisions for using frameworks like Netty and BFT-SMaRt to build the FIT-Broker. Also, our implementation of the proposed architecture is presented. It goes layer by layer detailing the functionalities of each component and presents the available interfaces for enhancement.

Since the FIT-Broker offers multiple levels of QoP and QoS, it becomes possible to create different services with different levels of fault tolerance. This means that clients may chose the level of fault tolerance that better suits them.

The evaluation of the FIT-Broker led to several conclusion. The first is that ordering protocols have a big impact in performance. The second is the communication represents a big part of the time it takes to interact with the FIT-Broker. The third is that we reach a maximum throughput between two and five publishers per channel. The fourth is that by batching we can achieve very high throughput values. In addition, we where able to conclude that there is small degradation to the service even in the presence of failures.

Additionally we have presented an application that serves as prove of concept of the FIT-Broker. It shows the client throughput and offers the means to inject faults that trigger the implemented protocols in the replicas. Also, it shows that clients continue to function properly has long as the number of injected faults is smaller than f and that when the number of faults passes f the clients stop.

We believe that the work presented in this thesis provides a significant contribution for the development of reliable publish-subscribe system. It proves that it is possible provide a reliable message routing service even in the presence of failures.

6.2 Future work

The evaluation made to the FIT-Broker showed a good performance in a local-are-network (LAN). It would be interesting to see and compare the degradation in performance if we executed the same tests in wide-are-networks (WAN).

Since we consider that the worst throughput is when the QoS is ordered and the QoP is BFT, we have only tested and compared for the cases where we set the QoP to BFT and vary the QoS. It would be interesting to set the QoP to CFT and perform the same tests to verify if the behaviour is maintained.

Another thing that is left as future work is the integration with monitoring systems. This can be done by making the probes publish their information in the FIT-Broker. Consoles should become subscribers and periodically consume the information from the output queues.

Appendix A

CFT ordering algorithm

Algorithm 6: CFT ordering algorithm (executed in the service replica)

```
1  Upon reception r = (REQUEST, seq, op)$a_c$ from c $\in$ $Clients$ do
2      requestsReceived($r$)

3  // Execute if leader
4  Upon $(toOrder \neq 0)$ and $(currentCons = -1)$ and $(\neg stopped)$ do
5      $Batch \leftarrow X \subseteq ToOrder : |X| \leq maxBatch$ and $fair(X)$
6      $currentCons \leftarrow$ highCons($DecLog$).$i + 1$
7      send(PROPOSE, $currentCons$, $Batch$) to $R$

8  Upon reception r = (PROPOSE, $i$, $Batch$) from $currentLeader$ do
9      $currentCons \leftarrow i$
10     foreach $r \in Batch$ do
11         cancelTimers($\{r\}$)
12         $ToOrder \leftarrow ToOrder / \{r\}$
13         $Proposed \leftarrow Proposed \bigcup \{r\}$
14     send(ACCEPT, $currentCons$) to $R$

15 Upon reception r = (ACCEPT, i) from $r^l \in R$ do
16     $Decided \leftarrow Decided \bigcup \{i\}$
17     if $Decided.size \geq f + 1$ and $currentCons \neq -1$ then
18         $currentCons \leftarrow -1$
19         $Decided \leftarrow 0$
20         foreach $r = \langle REQUEST, seq, op, c\rangle a_c \in Proposed$ do
21             $Proposed \leftarrow Proposed / \{r\}$
22             $rep \leftarrow$ execute($op$)
23             send$\langle REPLY, seq, rep\rangle$ to $c$
24         $DecLog \leftarrow DecLog \bigcup \{i\}$

25 Procedure RequestReceived(r)
26     if $lastSeq[c] + 1 = r.seq$ then
27         $ToOrder \leftarrow ToOrder \bigcup \{r\}$
28         activateTimers($\{r\}$, $timeout$)
29         $lastSeq[c] \leftarrow r.seq$
```

Appendix B

CFT leader election algorithm

Algorithm 7: CFT leader election (executed in the service replica)

```
1  Upon timeout for request $m \in M$ do
2      $stopped \leftarrow TRUE$
3      send(STOP, $creg$) to $R$
4  Upon reception of $\langle$STOP, $reg\rangle$ from $r^l \in R$ do
5      if $reg = creg$ then
6          if stopped = FALSE then
7              $stopped \leftarrow TRUE$
8          $creg \leftarrow creg + 1$
9          $leader \leftarrow creg \bmod n$
10         restartTimers($ToOrder$,$timeout$)
11         $stopped \leftarrow FALSE$
```

Appendix C

CFT state transfer protocol

Algorithm 8: CFT state transfer protocol (executed in the service replica)

```
Upon Init do
    $appStateOnly \leftarrow FALSE$
    $waitingEID \leftarrow -1$
    $completeState \leftarrow 0$
    $regencies[1..\infty]$
    $\forall n \in N_0 : regencies[n] \leftarrow -1$
    $views[1..\infty]$
    $\forall n \in N_0 : views[n] \leftarrow 0$
    $appStates[1..\infty]$
    $\forall n \in N_0 : appStates[n] \leftarrow 0$
Upon StateTimeout do
    changeStateReplica()
    requestState($waitingEID$)
Upon reception of $\langle ST\_REQUEST, EID, ID\rangle$ from $r^l \in R$ do
    $sendState \leftarrow (ID = r)$
    $appState \leftarrow$ getAppState($EID, sendState$) // Implemented by the Application
    send$\langle ST\_REPLY, EID, appState,$getCurrentView(),getCurrentRegency()$\rangle$ to $r^l$
Upon reception of $\langle ST\_REPLY, waitingEID, appState, view, regency\rangle$ do
    if $appStateOnly = FALSE$ then
        $regencies[waitingEID] \leftarrow regencies[waitingEID] \bigcup regency$
        $views[waitingEID] \leftarrow views[waitingEID] \bigcup view$
        $currentView \leftarrow view$
        $currentRegency \leftarrow regency$
    else
        $currentView =$ getCurrentView()
        $currentRegency =$ getCurrentRegency()
        $currentLeader =$ getCurrentLeader()
    $appStates[waitingEID] \leftarrow appStates[waitingEID] \bigcup appState$
    update($waitingEID, currentRegency, currentLeader, currentView, currentAppState$)
Procedure requestState(eid)
    $waitingEID \leftarrow eid$
    $completeState \leftarrow 0$
    stopMessageTimers()
    startStateTimeout()
    send⟨ ST_REQUEST,$waitingEID$,getStateReplica()⟩ to $R$
Procedure requestAppState(eid)
    $appStateOnly \leftarrow$ TRUE
    requestState($eid$)
Procedure update(eid,regency,leader,view,appState)
    setProtocolState($eid, regency, leader, view$)
    setAppState() // Implmented by the application
    $waitingEID \leftarrow -1$
    $completeState \leftarrow 0$
    if $appStateOnly = FALSE \wedge$ Leader change protocol was triggered then
        Move stopped messages to out of context
    Process out of context messages
    Restart timers for pending requests
    if $appStateOnly = TRUE$ then
        $appStateOnly \leftarrow FALSE$
        Resume leader change protocol
```

Appendix D

FIT-Broker interface

Listing D.1: Interface methods

```
/**
* client must issue this request to be allowed to publish in a channel
* registers a client to channel defined in TAG
* returns the result of the operation
*/
public Request register(TAG);

/**
* client must issue this request to be allowed to receive events from a channel
* subscribers a client to channel defined in TAG
* returns the result of the operation
*/
public Request  subscribe(TAG);

/**
* unregisters client from channel defined in TAG
* returns the result of the operation
*/
public Request  unregister(TAG);

/**
* unsubscribers client from channel defined in TAG
* returns the result of the operation
*/
public Request unsubscribe(TAG);

/**
* publishes event E in channel defined in TAG
* returns the result of the operation
*/
public Request  publish(TAG, E);

/**
* stores event E in local cache until a minimum number of events are collected
* publishes all events in cache in channel defined in TAG
* returns the result of the operation
*/
public Request publishWithCaching(TAG, E);
```

```

/**
* retrives a set of events from channel defined in TAG
* returns the result of the operation
*/
public Request poolEvent(TAG);

/**
* tries to retrive up to N events from channel defined in TAG
* returns the result of the operation
*/
public Request pollEventsFromChannel(TAG, N);

/**
* tries to retrieve up to the maximum number of events from channel defined in TAG
* returns the result of the operation
*/
public Request pollEventsFromChannelWithCaching (TAG)
```

Appendix E

Connection interface

Listing E.1: ClientSideConnection Interface

```
/**
* connectes to the service replicas
*/
public void start();

/**
* sends a request to the service replicas
* returns the result of the operation
*/
public Request invoke(R);

/**
* closes the connection to the service replicas
*/
public void close();
```

Appendix F

Service proxy methods

Listing F.1: Interface methods

```java
/**
* sends a request to the service replicas
* returns the result of the operation
*/
public Request invoke(R);

/**
* closes the connection to the service replicas
*/
public void closeConnection();

/**
* returns the number of events to be cached peer request
*/
public int getNumberOfEventsToCachePerRequest();

/**
* returns the maximum number of events to fetch peer request
*/
public int getMaxNumberOfEventsToFetchPerRequest();
```

Appendix G

Storage interface

Listing G.1: Data service methods

```
/**
* inserts channel ch into the date service
*/
public void insertChannel(CH);

/**
* inserts new publisher on channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean insertNewPublisher(PUB,TAG);

/**
* adds new subscriber on channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean insertNewSubscriber(SUB, TAG);

/**
* inserts event E on channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean insertNewEvent(E, TAG);

/**
* removes publisher with ID from channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean removePublisher(ID, TAG);

/**
* removes subscriber with ID from channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean removeSubscriber(ID, TAG);

/**
* removes event E from subscriber with ID on channel TAG
* returns true if operations is successful, false otherwise
```

```
*/
public boolean removeEventX(ID, E, TAG);

/**
* gets the next event for subscriber ID in channel TAG
* returns the next event
*/
public Event getNextEvent(TAG, ID);

/**
* verifies if channel exists
* returns true if channel exists, false otherwise
*/
public boolean hasChannel(String tag);

/**
* returns the total number of publishers for all channels
*/
public int getNumberOfPublishers();

/**
* returns the total number of subscribers for all channels
*/
public int getNumberOfSubscribers();

/**
* returns the number of publishers for channel TAG
*/
public int getNumberOfPublishersForChannel(TAG);

/**
* returns the number of subscribers for channel TAG
*/
public int getNumberOfSubscribersForChannel(TAG);

/**
* unregister publisher ID from all channels
* returns true if operations is successful, false otherwise
*/
public boolean unRegisterFromAllChannels(String id);

/**
* unrsubscribes subscriber ID from all channels
* returns true if operations is successful, false otherwise
*/
public boolean unSubscribeFromAllChannels(String id);

/**
* inserts a list of events EV on channel TAG
* returns true if operations is successful, false otherwise
*/
public boolean insertListOfEvents(EV, TAG);

/**
* gets N events from channel TAG for subscriber ID
* a list of events
```

```
*/
public ArrayList<Event> getEventsFromChannel(ID, TAG, N);
```

Appendix H

Channel interface

Listing H.1: Channel interface

```java
/**
* sets the next event id within the channel
*/
public void setNextIdWithinTheChannel(ID);

/**
* returns the TAG of the channel
*/
public String getTag();

/**
*
* returns the QoP of the channel
*/
public QoP getQoP();

/**
*
* returns the QoS of the channel
*/
public QoS getQoS();

/**
* adds publisher PUB to the channel
* returns true if operations is successful, false otherwise
*/
public boolean addPublisher(PUB);

/**
* removes publisher with ID from the channel
* returns true if operations is successful, false otherwise
*/
public boolean removePublisher(ID);

/**
* adds subscriber SUB to the channel
* returns true if operations is successful, false otherwise
*/
```

```
public boolean addSubscriber(SUB);

/**
 * removes subscriber with ID from the channel
 * returns true if operations is successful, false otherwise
 */
public boolean removeSubscriber(ID);

/**
* adds event E to the output queues of subscribers
* returns true if operations is successful, false otherwise
*/
public boolean addEventToSubscribers(E);

/**
* removes event E from subscriber with ID
* returns true if operations is successful, false otherwise
*/
public boolean removeEvent(ID, E);

/**
* returns the number of publishers in the channel
*/
public int getNumberOfPublishers();


/**
* returns the number ofsubscribersin the channel
*/
public int getNumberOfSubscribers();

/**
 * adds list of events EV to the output queue of subscribers
 * returns true if operations is successful, false otherwise
 */
public boolean addListOfEventsToSubscribers(EV);

/**
* gets the next event in the output queue of subscriber with ID
* returns the next event
*/
public Event getNextEvent(ID);

/**
* tries to fetch up to N events from the output queue of subscriber with ID
* returns the list of events
*/
public ArrayList<Event> getEventsForSubscriber(ID, N;
```

Bibliography

[1] Amazon. Amazon cloudwatch, 2012. http://aws.amazon.com/cloudwatch/.

[2] Roberto Baldoni, Leonardo Querzoni, and Antonino Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, 2005.

[3] Alysson Neves Bessani. From byzantine fault tolerance to intrusion tolerance (a position paper). In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 15–18, Washington, DC, USA, 2011. IEEE Computer Society.

[4] Bessani, A. N.; et. al. bft-smart - high-performance byzantine-fault-tolerant state machine replication, 2011. http://code.google.com/p/bft-smart/.

[5] Roy H. Campbell, Mirko Montanari, and Reza Farivar. A middleware for assured clouds. *Journal of Internet Services and Applications*, 12/2011 2011.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[7] Rackspace Cloudkick. Cloudkick - cloud management, 2012. https://www.cloudkick.com.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10:642–657, June 1999.

[9] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[10] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[11] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, May 2001.

[12] Ethan Galstad. Nagios, 2013. http://www.nagios.org/.

[13] Xiangfeng Guo, Jun Wei, and Dongli Han. Efficient event matching in publish/subscribe: Based on routing destination and matching history. In *Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*, NAS '08, pages 129–136, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Andreas Haeberlen, Petr Kouznetsov, and Peter 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.

[15] HP. ArcSight, 2012. http://www.arcsight.com/products/products-esm/.

[16] Diego Kreutz, António Casimiro, João Sousa, Alysson Bessani, and Igor Antunes. TRONE project - Deliverable D11: First specification of the communication protocols and middleware. Technical report, Faculty of Sciences of University of Lisbon, nov 2012. http://trone.di.fc.ul.pt.

[17] Leslie Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978.

[18] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[20] Butler Lampson. The ABCD of Paxos. In *Proceedings of the twentieth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 2001.

[21] Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical report, Computer Science Department, Indiana University, 2003.

[22] LogicMonitor. Logicmonitor - hosted monitoring of network, servers, applications, stor- age, and cloud, 2012. http://www.logicmonitor.com/.

[23] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):202 –215, july-sept. 2006.

[24] Monitis. Monitis all-in-one monitoring platform - monitor everything, 2012. http://portal.monitis.com/.

[25] Liam O. Nicolas Falliere. W32.Stuxnet Dossier, 2011.

[26] Oracle. Java, 2013. https://www.java.com.

[27] Smruti Padhy, Diego Kreutz, António Casimiro, and Marcelo Pasin. TRONE project - Deliverable D10: First Specification of the Architecture. Technical report, Faculty of Sciences of University of Lisbon, oct 2011. http://trone.di.fc.ul.pt.

[28] The Netty project. Netty, 2013. https://www.netty.io.

[29] O. Ru andtti, Z. Milosevic, and A. Schiper. Generic construction of consensus algorithms for benign and byzantine faults. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 343 –352, 28 2010-july 1 2010.

[30] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[31] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101 –102, aug 2001.

[32] Dawn Xiaodong Song and Jonathan K. Millen. Secure auctions in a publish/subscribe system, 2000.

[33] J. Sousa and A. Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 37 –48, may 2012.

[34] J. Spring. Monitoring cloud computing by layer, part 1. *Security Privacy, IEEE*, 9(2):66 –68, march-april 2011.

[35] CA Technologies. Nimsoft monitor, 2012. http://www.nimsoft.com/solutions/.

[36] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677. 2003.

[37] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[38] Dinesh Chandra Verma. *Principles of Computer Systems and Network Management*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[39] Vmware. Vmware vfabric hyperic, 2012. http://www.vmware.com/products/vfabric-hyperic/.

[40] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.

[41] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, October 2003.

[42] Zenoss. Zenoss - the cloud management company, 2012. http://www.zenoss.com/.

[43] Piotr Zieliński. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory, June 2004.