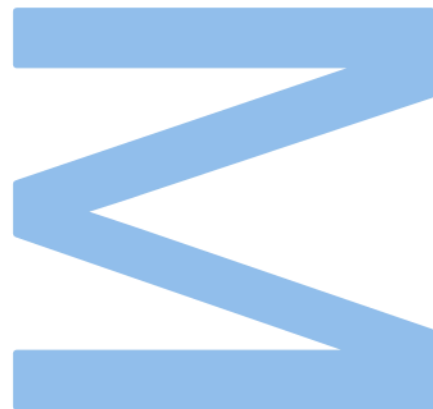


# Highly available and Fault Tolerant IoT Architectures



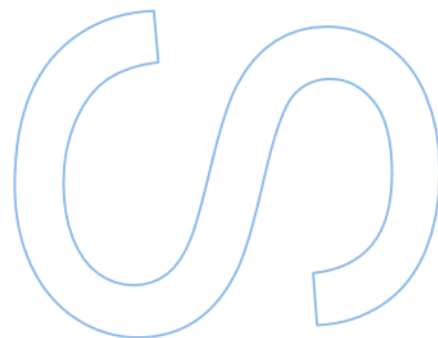
**Francisco Miguel Andrês Ribeiro**

Master in Information Security

Department of Computer Science

Faculty of Science of the University of Porto

2025



# Highly available and Fault Tolerant IoT Architectures

**Francisco Miguel Andrês Ribeiro**

Dissertation carried out as part of the Master in Information  
Security

Department of Computer Science  
2025

**Supervisor**

João Paulo da Conceição Soares, PhD, Faculty of Sciences

# Acknowledgements

I would like to begin by expressing my gratitude to my supervisor, João Soares, for allowing me to work on this project. His guidance, expertise, and constructive feedback helped shape this work, providing valuable input at every step of the dissertation.

To my closest family: my mother, father, grandmother, and especially my sister, who have always supported me throughout my personal and academic journey, allowing me to reach my goals. They provided me with the drive to always seek more and work harder. Thank you for helping me overcome all my struggles and for continuing to motivate me to follow my dreams and reach my objectives.

To my close friends, whom I've shared and continue to share many moments of my life, and whom I will remember forever. Frequently, they were my first line of support when I faced difficulties. They were always keen to assist me and make sure they were there to help me. I would particularly like to point out some of them: Rui Jorge, Diogo Vale, João Alves, Rita Romani, Jaime Silva, Duarte Alves, Manuel Sá, and Matilde Barbosa. I sincerely thank you for your invaluable support, endless stories, and memories, and for shaping my life into what it is today. Thank you for teaching me many life-shaping lessons, for being present when it matters, and for allowing me to enter your lives and have the pleasure of also supporting you in achieving your dreams.

Lastly, I would like to mention again my sister, who is the person I most value and who was my ultimate support through all the difficulties we've shared growing up. I dedicate this work to her as a token of my deepest respect, appreciation, and gratitude. Thank you for your care and for teaching me to always seek to make the difficult but correct decisions when it matters most.

# Resumo

O rápido crescimento de sistemas Internet of Things (IoT) tem levado à integração de dispositivos em domínios de missão crítica, tornando a fiabilidade e a segurança destes sistemas aspetos fundamentais. As arquiteturas tradicionais de IoT, frequentemente baseadas no modelo Publish/Subscribe (Pub/Sub), revelam-se vulneráveis a faltas, em particular as não benignas, em que componentes podem exibir comportamentos arbitrários ou maliciosos. A existência de pontos únicos de falha, como brokers não replicados, compromete a integridade do sistema, uma vez que não é possível garantir que os dados não sejam perdidos, duplicados ou adulterados. Esta dissertação aborda estes desafios através da conceção, implementação e validação de uma arquitetura multi-camada que integra tolerância a faltas bizantinas Byzantine Fault Tolerance (BFT) em sistemas IoT, garantindo a consistência e a disponibilidade dos dados mesmo na presença de componentes maliciosos.

A arquitetura proposta visa aumentar a resiliência do sistema, procurando simultaneamente mitigar os impactos da replicação de componentes. Uma das principais contribuições consiste na redução da replicação ao nível dos sensores e dos clientes bizantinos, inspirando-se em soluções para o problema de Sensor Fusion, diminuindo assim os custos e a sobrecarga de rede sem comprometer as propriedades bizantinas. Para mitigar o risco de propagação de falhas entre camadas - uma vulnerabilidade significativa em arquiteturas BFT multicamada - é adotada uma estratégia em que os sensores enviam as suas leituras diretamente para todos os clientes bizantinos. Estes, por sua vez, realizam um passo de agregação que permite mascarar valores incorretos previamente a invocar o consenso. Além disso, são incorporados metadados criptográficos como prova de consenso — conjunto maioritário de assinaturas e um contador monotónico — diretamente nos dados. Esta abordagem permite que as aplicações verifiquem a autenticidade e a atualidade das notificações provenientes de brokers, prevenindo ataques de repetição (replay attacks) e assegurando a correta execução do sistema.

A eficácia e eficiência da arquitetura proposta foram validadas através de uma implementação prática e de um conjunto de experiências comparativas com um sistema de base não replicado. Apesar de ser provado que a introdução de mecanismos de replicação e tolerância a faltas bizantinas aumentam a resiliência do sistema, é demonstrado que existe um custo mensurável: os resultados evidenciam que os mecanismos BFT introduzem uma latência significativa quando comparados com arquiteturas não replicadas.



# Abstract

The rapid growth of the Internet of Things (IoT) has interconnected devices become part of mission-critical domains, making the reliability and security of these systems paramount. Traditional IoT architectures, often based on a Publish/Subscribe (Pub/Sub) model, are vulnerable to failures, particularly non-benign ones where components exhibit arbitrary or malicious behavior. The presence of a single point of failure, such as non-replicated brokers, can also compromise system integrity, as it cannot be guaranteed that data has not been dropped, duplicated, or altered. This thesis addresses these challenges by designing, implementing, and evaluating a multi-layered architecture that integrates Byzantine Fault Tolerance (BFT) into an IoT Pub/Sub system to ensure data consistency and availability even in the presence of malicious nodes.

This work proposes an architecture that enhances resilience while trying to reduce the impacts of component replication. A key contribution is the reduction of replication at the sensor and byzantine clients layer, drawing inspiration from solutions to the Sensor Fusion problem, which decreases costs and network overhead without compromising Byzantine properties. To counter the risk of fault propagation across layers, a significant vulnerability in naive multi-layer BFT systems, the architecture mandates that sensors multicast data to clients. This allows clients to perform an aggregation step, masking erroneous values before invoking consensus. Furthermore, the system eliminates the need for a second BFT consensus layer for notification validation by embedding cryptographic proof-of-consensus metadata - specifically, quorum signatures and monotonic counters - into the data itself. This enables applications to efficiently verify the authenticity and freshness of broker notifications, preventing replay attacks and ensuring correctness.

The effectiveness of the proposed architecture was validated through a practical implementation and a series of experiments comparing its performance against a non-replicated baseline system. The evaluation, conducted under various workloads and simulated attack scenarios, demonstrates that the introduction of replication and Byzantine fault-tolerant mechanisms successfully enhances fault tolerance and system resilience. However, this increased reliability comes at a quantifiable cost. The results show that the BFT mechanisms introduce significant latency and reduce overall throughput when compared to the baseline non-replicated architectures.

# Table of Contents

|                                                                                                         |      |
|---------------------------------------------------------------------------------------------------------|------|
| List of Tables .....                                                                                    | viii |
| List of Figures .....                                                                                   | ix   |
| Acronyms .....                                                                                          | xi   |
| 1. Introduction .....                                                                                   | 1    |
| 1.1. Organization .....                                                                                 | 2    |
| 2. Background .....                                                                                     | 4    |
| 2.1. Inherent Challenges in IoT Systems .....                                                           | 4    |
| 2.1.1. Pub/Sub Model of IoT .....                                                                       | 4    |
| 2.2. Fault Tolerance .....                                                                              | 5    |
| 2.2.1. Crash Fault Tolerance .....                                                                      | 6    |
| 2.2.2. Byzantine Generals Problem .....                                                                 | 6    |
| 2.2.3. Byzantine Fault Tolerance .....                                                                  | 8    |
| 2.3. Frameworks .....                                                                                   | 10   |
| 2.3.1. FIWARE .....                                                                                     | 10   |
| 2.3.2. BFT-SMaRt .....                                                                                  | 11   |
| 2.4. Summary .....                                                                                      | 12   |
| 3. State of the Art .....                                                                               | 13   |
| 3.1. Fault-Tolerant Sensor Readings .....                                                               | 13   |
| 3.1.1. Sensor Fusion Problem .....                                                                      | 13   |
| 3.1.2. Approximate Agreement .....                                                                      | 15   |
| 3.1.3. Inexact Agreement .....                                                                          | 16   |
| 3.1.4. Brooks–lyengar Algorithm .....                                                                   | 17   |
| 3.2. Byzantine Fault Tolerance in Publish/Subscribe systems .....                                       | 18   |
| 3.2.1. Efficient Middleware for Byzantine Fault Tolerant Database Replication<br>[31] - Byzantium ..... | 19   |
| 3.2.2. Byzantine Fault-Tolerant Publish/Subscribe: A Cloud Computing Infras-<br>tructure .....          | 20   |
| 3.2.3. Trinity [33] .....                                                                               | 21   |
| 3.2.4. Galaxy [34] .....                                                                                | 22   |
| 3.3. Summary .....                                                                                      | 24   |

|        |                                                             |    |
|--------|-------------------------------------------------------------|----|
| 4.     | Architecture .....                                          | 25 |
| 4.1.   | IoT Architectures Under Faults .....                        | 25 |
| 4.1.1. | Non-replicated Pub/Sub Architectures .....                  | 25 |
| 4.1.2. | Introducing Broker Replication .....                        | 27 |
| 4.2.   | Naive Approach .....                                        | 29 |
| 4.2.1. | Architecture Concept .....                                  | 29 |
| 4.2.2. | Middleware-Based Architecture .....                         | 32 |
| 4.2.3. | Fault Model .....                                           | 34 |
| 4.3.   | Proposed Architecture .....                                 | 38 |
| 4.3.1. | System Flow .....                                           | 39 |
| 4.3.2. | Attacker/Fault Model .....                                  | 41 |
| 4.3.3. | Further Assumptions .....                                   | 43 |
| 4.4.   | Summary .....                                               | 44 |
| 5.     | Implementation .....                                        | 45 |
| 5.1.   | Implementation of the Byzantine Middleware Components ..... | 45 |
| 5.1.1. | BFT Broker Proxy .....                                      | 46 |
| 5.1.2. | BFT Broker Client .....                                     | 47 |
| 5.2.   | Application-level code .....                                | 50 |
| 5.2.1. | Airquality Sensor Application .....                         | 50 |
| 5.2.2. | Application .....                                           | 51 |
| 5.3.   | Summary .....                                               | 51 |
| 6.     | Results .....                                               | 53 |
| 6.1.   | Experimental Setup .....                                    | 53 |
| 6.1.1. | System Environment .....                                    | 53 |
| 6.1.2. | Data Collection and Metrics Extraction .....                | 53 |
| 6.1.3. | Workload and Scenarios .....                                | 54 |
| 6.2.   | Non-replicated system .....                                 | 55 |
| 6.2.1. | Sensor(Publisher) flow .....                                | 55 |
| 6.2.2. | End-to-End flow .....                                       | 56 |
| 6.3.   | Replicated system: No Faults .....                          | 57 |
| 6.3.1. | Sensor(Publisher) flow .....                                | 57 |
| 6.3.2. | End-to-End flow .....                                       | 58 |
| 6.4.   | Attacker Simulations .....                                  | 60 |
| 6.5.   | Summary .....                                               | 62 |
| 7.     | Conclusion .....                                            | 64 |
| 7.1.   | Implemented system .....                                    | 64 |
| 7.2.   | Limitations .....                                           | 65 |

|                                         |    |
|-----------------------------------------|----|
| 7.3. Future Work and Improvements ..... | 65 |
| Bibliography .....                      | 66 |

# List of Tables

|                                       |    |
|---------------------------------------|----|
| 4.1. Network message complexity ..... | 35 |
|---------------------------------------|----|

# List of Figures

|                                                                                                             |    |
|-------------------------------------------------------------------------------------------------------------|----|
| 2.1. Commander as traitor in a $2f + 1$ system [5].....                                                     | 7  |
| 2.2. Lieutenant as traitor in a $2f + 1$ system [5].....                                                    | 7  |
| 2.3. Commander as traitor detectable in a $2f + 1$ system but no majority agreement can be reached[5] ..... | 8  |
| 2.4. PBFT example of consensus phases [15] .....                                                            | 9  |
| 2.5. Quorum intersection with $3f + 1$ replicas [19].....                                                   | 9  |
| 2.6. Simplistic overview of FIWARE's architecture .....                                                     | 11 |
| 2.7. BFT-SMaRt's normal message exchange pattern[16].....                                                   | 12 |
| 3.1. Marzullo's fusion interval according to the number of tolerated faults [25] .....                      | 14 |
| 3.2. Marzullo's fusion interval inaccuracy problem if the majority of sensors are faulty [26].....          | 15 |
| 3.3. Asynchronous approximation algorithm.[27].....                                                         | 16 |
| 3.4. Fast Convergence algorithm - process execution.[28] .....                                              | 17 |
| 3.5. Dart short analogy to explain Precision versus Accuracy concepts.[29].....                             | 18 |
| 3.6. Byzantium system architecture.[31] .....                                                               | 20 |
| 3.7. Trinity system architecture. [33].....                                                                 | 22 |
| 3.8. Galaxy's system architecture. [34].....                                                                | 23 |
| 4.1. Simplified depiction of Pub/Sub systems' architecture[35] .....                                        | 26 |
| 4.2. Kafka architecture[37] .....                                                                           | 28 |
| 4.3. Naive Architecture .....                                                                               | 30 |
| 4.4. Message model for Entity Creation and Update .....                                                     | 31 |
| 4.5. Notifications Message Model .....                                                                      | 31 |
| 4.6. Application Subscription and Query Message model .....                                                 | 31 |
| 4.7. Middleware-based Architecture model.....                                                               | 32 |
| 4.8. Fault propagation scenario 1 .....                                                                     | 36 |
| 4.9. Fault propagation scenario 2.....                                                                      | 37 |
| 4.10Simplified system layout of publishing end-to-end flow .....                                            | 38 |
| 4.11Refined Architecture .....                                                                              | 39 |
| 4.12Client Aggregation scenario .....                                                                       | 40 |
| 4.13Fault propagation mitigation.....                                                                       | 42 |
| 4.14Full Architecture Worst Case mitigation.....                                                            | 42 |
| 5.1. Previous Notification Message flow .....                                                               | 49 |
| 5.2. New Notification Message flow .....                                                                    | 50 |
| 6.1. Experimental Setup Diagram .....                                                                       | 54 |
| 6.2. Average Throughput over time.....                                                                      | 55 |
| 6.3. Average Latency over time .....                                                                        | 55 |
| 6.4. Average Packet Count over time .....                                                                   | 55 |

|                                        |    |
|----------------------------------------|----|
| 6.5. Average Bytes over time .....     | 55 |
| 6.6. Average Throughput over time..... | 56 |
| 6.7. Average Latency over time .....   | 56 |
| 6.8. Packet Count over time .....      | 56 |
| 6.9. Amount of Bytes over time .....   | 56 |
| 6.10Average Throughput over Time ..... | 57 |
| 6.11Latency over Time .....            | 57 |
| 6.12Average Packet Count .....         | 57 |
| 6.13Average Bytes per Second .....     | 57 |
| 6.14Average Throughput over Time ..... | 58 |
| 6.15Average Latency over Time .....    | 58 |
| 6.16Average Packet Count .....         | 59 |
| 6.17Average Bytes per Second .....     | 59 |
| 6.18Average Throughput over Time ..... | 60 |
| 6.19Latency over Time .....            | 61 |
| 6.20Average Packet Count .....         | 61 |
| 6.21Average Bytes per Second .....     | 61 |

# Acronyms

**ABA** Approximate Byzantine Agreement

**API** Application Programming Interface

**BFT** Byzantine Fault Tolerance

**BFT-IoT** Byzantine Fault Tolerance in Internet of Things

**BFT-PB** Byzantine Fault Tolerance in Publish/Subscribe

**BGP** Byzantine Generals Problem

**CFT** Crash Fault Tolerance

**CWA** Crash and Withholding Avoidance

**DDoS** Distributed Denial of Service

**EMA** Exponential Moving Average

**FCA** Fast Convergence Algorithm

**GST** Global Stabilization Time

**IoT** Internet of Things

**MQTT** Message Queuing Telemetry Transport

**NGSI-LD** Next Generation Service Interface with Linked Data

**PBFT** Practical Byzantine Fault Tolerance

**Pub/Sub** Publish/Subscribe

**QoS** Quality of Service

**SI** Snapshot Isolation

**SMR** State Machine Replication

**SPoF** Single Points of Failure

# Chapter 1

## Introduction

The Internet of Things (IoT) represents a paradigm shift in computing, weaving connectivity and data processing capabilities into our physical world. Interconnected devices, ranging from simple sensors and actuators to complex edge gateways, are generating unprecedented volumes of data and enabling transformative applications across diverse sectors, including smart homes, industrial automation, healthcare, smart cities, and environmental monitoring [1]. As these networks expand, ensuring reliable operation and consistent data availability becomes an important challenge, particularly in mission-critical domains, where even minor disruptions can have severe consequences [2].

The very characteristics that define IoT systems - massive scale, device heterogeneity, resource constraints (limited power, computation, memory), and often deployed in physically inaccessible or unattended environments introduce significant challenges. Foremost among these are concerns regarding security and reliability. Traditional security mechanisms often prove inadequate for constrained devices, leaving systems vulnerable to attacks [3]. The distributed and often complex nature of these systems makes them susceptible to failures, ranging from simple node crashes, caused by hardware malfunction or power loss, to more insidious Byzantine failures [4].

Distributed systems must cope not only with benign failures, such as crashes or temporary unresponsiveness, but also with arbitrary or even coordinated malicious behaviors known as Byzantine faults. The Byzantine Generals Problem (BGP) was originally formalized by Lamport, Shostak, and Pease [5] and later led to practical consensus protocols such as Practical Byzantine Fault Tolerance (PBFT) [6]. A Byzantine Fault Tolerance (BFT) system is designed to guarantee two essential properties:

1. **Safety** (Consistency): All correct (non-faulty) nodes agree on the same state or value even if up to  $f$  nodes behave arbitrarily or maliciously.
2. **Liveness** (Availability): The system eventually reaches a decision and continues to operate despite the presence of up to  $f$  faulty nodes.

Most protocols addressing Byzantine faults require a minimum of  $3f + 1$  nodes to tolerate up to  $f$  Byzantine failures [5]. This traditional requirement ensures that even if up to  $f$  nodes are compromised, the remaining  $2f + 1$  honest nodes can form a majority quorum to reach agreement. However, in IoT environments, such high levels of replication may increase costs, energy consumption, and network congestion.

The central objective of this dissertation is to investigate the feasibility of reducing the amount of deployed sensors while maintaining the number of consensus replicas thus preserving safety and liveness properties of Byzantine fault-tolerant protocols. More specifically, this work examines a model that employs only  $2f + 1$  sensors (with  $f$  representing the maximum number of tolerable faulty sensors, clients and replicas) to deliver aggregated sensor data. As aggregated data is then transmitted to a set of consensus replicas, it potentially reduces redundancy without compromising byzantine properties. This research is motivated by the following concerns:

- **Cost Efficiency:** Fewer sensors lead to a reduction in installation and operational costs.
- **Energy Savings:** Lower sensor redundancy results in reduced energy consumption, which is especially crucial for energy-constrained IoT devices.
- **Network and Processing Overhead:** Minimizing sensor transmissions can ease network bandwidth requirements and reduce processing load on byzantine consensus nodes.

## 1.1 Organization

The remainder of this dissertation is structured to progressively build upon foundational concepts, leading to the practical implementation and evaluation of the proposed system. Chapter 2 provides a comprehensive overview of essential concepts necessary to understand the context of this work. It delves into Byzantine Fault Tolerance(BFT), exploring

how systems can maintain reliability despite arbitrary faults, and examines properties such as quorum intersection and other critical aspects that ensure consistency in consensus algorithms. Additional relevant topics are also discussed to establish a solid theoretical foundation. Building upon this, Chapter 3 surveys current state-of-the-art solutions related to Byzantine Fault Tolerance in Internet of Things (BFT-IoT) systems. It analyzes existing methodologies, highlighting their strengths and limitations, and identifies areas where the proposed solution offers improvements or novel approaches. Chapter 4 outlines the theoretical framework developed to achieve the objectives of this research. It details the system architecture, defines the roles and interactions of various components, and includes formal arguments to demonstrate the feasibility and correctness of the proposed approach. Following this, Chapter 5 discusses the practical implementation of the system. It describes the development process, including technology choices, coding practices, and any adjustments or optimizations made to address any constraints and enhance system performance. Chapter 6 presents the results obtained from testing the implemented system. It presents an analysis of the cost associated with adding replication and fault tolerance mechanisms, and an evaluation of attack scenarios. Finally, Chapter 7 concludes the dissertation by summarizing the key findings and contributions of the research. It reflects on the implications of the work, discusses its limitations, and suggests directions for future research and development to further advance the study of BFT-IoT systems.

## Chapter 2

# Background

This chapter delves into foundational concepts pertinent to this dissertation, by exploring further on topics such **IoT** and **BFT**, to establish the essential building blocks and provide more context in the objectives of this study.

### 2.1 Inherent Challenges in IoT Systems

The rapid expansion of **IoT** has introduced significant challenges that impact the security and reliability of these systems. The diverse and resource-constrained nature of **IoT** devices, coupled with their use in mission-critical infrastructures, increases the necessity to address challenges in ensuring the secure functioning of these deployments [7].

Resource limitations often prevent the implementation of advanced security protocols, making these devices more susceptible to breaches. Yu, Zhang, and Wang [7] highlight that the inherent heterogeneity and resource constraints of **IoT** devices necessitate tailored security solutions to mitigate vulnerabilities effectively. Ideally, these systems must be deployed in a fault-resilient manner to handle hardware malfunctions, network disruptions, or software errors.

#### 2.1.1 Pub/Sub Model of IoT

The **Pub/Sub** communication model facilitates efficient data exchange within **IoT** systems. It allows messages to be disseminated to specific topics without direct knowledge of subscribers, who, in turn, receive only messages of interest based on their subscriptions(notifications). This decoupling enhances scalability and flexibility, accommodating

the dynamic nature of IoT environments. Another key aspect is the support for asynchronous communication, allowing devices to operate independently and efficiently manage network resources. Protocols like Message Queuing Telemetry Transport (MQTT) are commonly employed to implement this model, offering lightweight messaging suitable for resource-constrained environments [8].

Integrating this model into IoT architectures not only streamlines communication but also enhances the system's ability to handle scalability demands. This is particularly relevant when developing BFT-IoT systems, where robust and lightweight communication mechanisms are crucial to guarantee the system's reliability and integrity. The overhead already imposed by the replication methodologies of BFT systems requires the underlying Pub/Sub system to be optimized for performance.

## 2.2 Fault Tolerance

According to Kamil et al.[9], fault tolerance refers to *"The capacity of a system to function commonly, even if one or more of its elements fails"*. The objective of fault-tolerant systems is to provide a continuous flow of execution, assuming some components might fail. This could result in higher availability and also in an improvement in Quality of Service (QoS). As explained, redundancy is one of the techniques that enables systems to withstand faults. The idea is to replicate entities in a system - network, hardware, or software related - so that if one fails the remaining components can keep the whole system running. The authors also reference that from a *"Fault Tolerance perspective, a reliable system cannot rely on any one part performing as intended"*[9], meaning that to achieve fault tolerant systems there shouldn't be any Single Points of Failure (SPoF) - agents that when disrupted could cause the whole system to stop working as intended. In any case, redundancy raises operational costs. This is reflected in the increased complexity to maintain the system as well as the potential requirement for either additional or more capable hardware. As a result, this technique is mostly used in situations where its advantages exceed the disadvantages of cost and/or mission-critical scenarios[10] where the potential malfunction of the system could incur great loss.

The next subsections explore further the topic of fault tolerance, giving more context into different levels of fault tolerance. It will also introduce one of the main concepts relevant to this dissertation, which the full body of work is based upon.

### 2.2.1 Crash Fault Tolerance

Crash Fault Tolerance (CFT) is the ability of a system to continue normal execution even in the case of failure of some of its components. It assumes a bounded number of components(replicas) could fail during execution due to software, hardware, or network failures, which can cause an unexpected "crash". Hadzilacos [11] defines a hierarchy of faults that range from the most benign to the most critical, which is later referenced in Bazzi et al. [12] as follows:

1. **Crash Failures:** caused by processors that behave correctly until they are prematurely halted and take no further actions.
2. **Send-Omission Failures:** caused by processors that stop or intermittently omit/fail to send some messages during execution. Remaining received or sent messages are always correct.
3. **General Omission Failures:** caused by processors that stop or intermittently fail to send or receive messages. Remaining received or sent messages are always correct.
4. **Arbitrary Failures:** caused by processors that can make arbitrary state transitions and send arbitrary messages. All messages can be incorrect or non-existent (Byzantine faults).

Typically, these systems use a quorum-based strategy where all replicas exchange received information and form a decision based on the majority of values read/received. As replicas are considered benign - can't maliciously send arbitrary messages - a CFT system can withstand up to  $f$  crash-faulty replicas in a  $2f + 1$  quorum, meaning there are  $f + 1$  not-faulty ones(simple majority).

### 2.2.2 Byzantine Generals Problem

Building upon fault-tolerant distributed systems, the Byzantine Generals Problem(BGP) explores the most challenging of fault types: Byzantine faults - characterized by arbitrary or malicious behavior of system components. Lamport et al. [5] demonstrate the problem with a group of lieutenants and one commander that instruments them to a simple decision - to attack or retreat. The decision making follows 2 conditions described by Lamport as *Interactive Consistency*[5]:

1. All loyal lieutenants obey the same order.
2. If the general is loyal, then every loyal lieutenant obeys the order he sends.

Lamport argues that if a general is loyal, then all loyal lieutenants follow their decision, and so the decision passed by the general should be taken by all of them. However, if not loyal, lieutenants can receive different messages (Figure 2.1), thus not following the first condition.

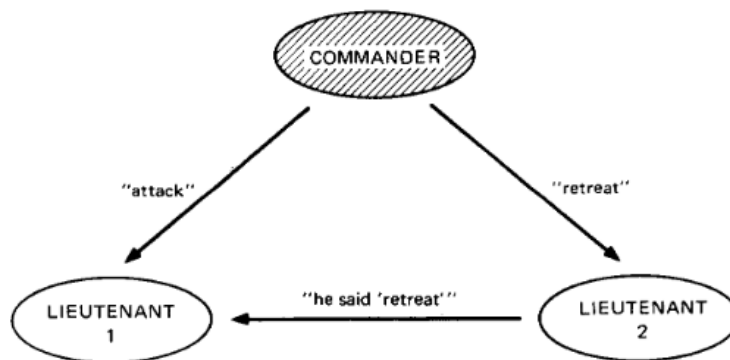


Figure 2.1: Commander as traitor in a  $2f + 1$  system [5]

Also, as shown by Figure 2.2, even if a commander is loyal and sends consistent messages, a malicious lieutenant can still try to "poison" loyal ones.

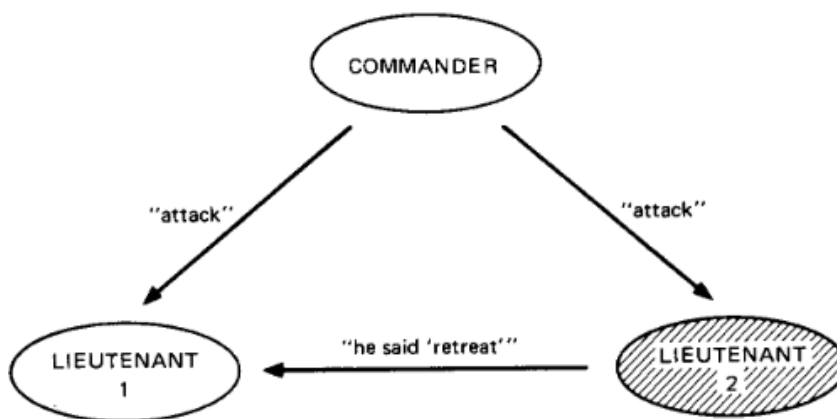


Figure 2.2: Lieutenant as traitor in a  $2f + 1$  system [5]

As a result, Lieutenant 1 cannot distinguish who the traitor is between Commander and Lieutenant 2.

Lamport[5] adds that this problem can be solved by having signed messages, assuming signatures cannot be forged. However, this only helps lieutenants to understand that

the commander is malicious. If, for example, we imply that a decision requires a majority of agreeing values, there is no longer the possibility of achieving this having just 3 generals (Figure 2.3).

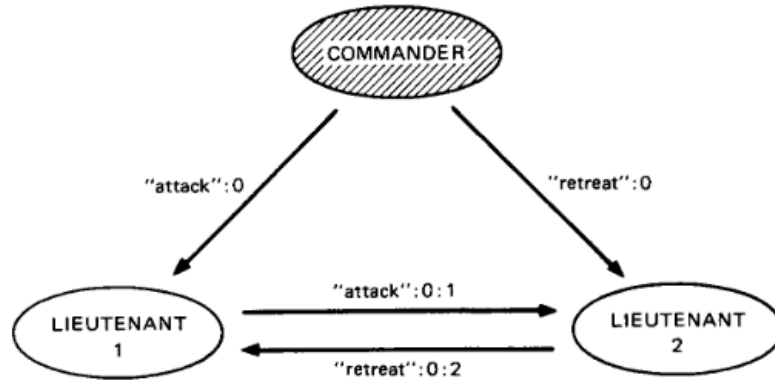


Figure 2.3: Commander as traitor detectable in a  $2f + 1$  system but no majority agreement can be reached[5]

## 2.2.3 Byzantine Fault Tolerance

**BFT** further improves the notion of fault-tolerant systems by dealing with arbitrary faults. It assumes nodes might send conflicting messages to different parts of the system, pretending to be functioning correctly while subtly corrupting data, or colluding with other malicious nodes. This makes **BFT** significantly more challenging than **CFT**. A simple majority is insufficient as Byzantine nodes can lie about their votes, making the system as a whole inconsistent, hence the typical  $3f + 1$  nodes required assuming  $f$  faulty ones [5, 6].

### 2.2.3.1 State Machine Replication

State Machine Replication (**SMR**) is a technique used in **BFT** protocols for distributed systems through replicating state across multiple nodes. By replicating the state, it ensures that if some bounded number of servers( $f$ ) fail, the service remains available. Schneider describes the idea that non-faulty **SMR** replicas see the same requests and process them in the same order, providing consistent executions(replica coordination)[13]. Also, replicas transition through the same sequence of states and can perform state synchronization if one becomes corrupted[13].

### 2.2.3.2 Consensus

Consensus is the process by which nodes can agree upon a sequence of operations even in the presence of arbitrary(byzantine) behavior. As shown in Figure 2.4, **BFT** protocols

rely on a multi-phase process where nodes exchange received data until a certain threshold is reached. After that, nodes move to the next phase, therefore altering their state. The principle here is to neutralize the effect of malicious nodes that try to deceive other replicas by multicasting corrupted/invalid data to all participants. This ensures all nodes can see a majority decision before committing the result[6, 14–16].

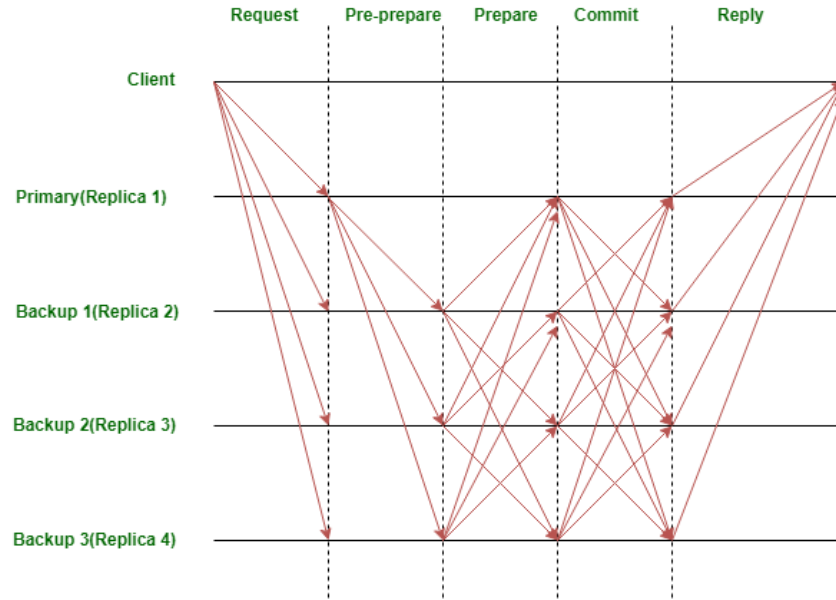


Figure 2.4: PBFT example of consensus phases [15]

### 2.2.3.3 Quorum Intersection

Quorum Intersection is a property of BFT protocols. In  $3f + 1$  replica systems, any two quorums of  $2f + 1$  nodes intersect in at least 1 non-faulty node[17]. For every possible quorum combination of  $2f + 1$  nodes, there is always an intersection of at least  $f + 1$  nodes, thus at least one non-faulty (Figure 2.5). This effectively blocks a scenario where 2 disjoint quorums simultaneously decide on different results, as at least one non-faulty replica is present in all phases of the consensus, therefore avoiding inconsistencies and guaranteeing **safety**[18, 19].



Figure 2.5: Quorum intersection with  $3f + 1$  replicas [19]

## 2.3 Frameworks

This section gives a brief overview of the frameworks used in the development of the practical solution to achieve a **BFT-IoT** system. It focuses on describing each framework with emphasis on the aspects most relevant to the dissertation.

### 2.3.1 FIWARE

FIWARE is an open-source framework that provides a set of standards and Application Programming Interfaces (**APIs**) that simplify the development of smart solutions, particularly in the context of the **IoT**, Smart Cities, and data-driven applications[20]. Its functionality is focused on context information by providing tools to standardize, manage, and extract information from [21]. It uses open standards and promotes interoperability, allowing developers to combine different technologies and services seamlessly.

#### 2.3.1.1 Context Information Management

A core concept in FIWARE is the management of context information, which represents real-world entities and their relationships. This is handled by the **Orion Context Broker**, a central component responsible for storing, managing, and providing access to context data[22].

#### 2.3.1.2 NGSI-LD

Next Generation Service Interface with Linked Data (**NGSI-LD**) governs the interaction with context information. It provides **APIs** that allows applications to access and manipulate data held by the Orion Context Broker consistently. NGSI-LD supports operations including creating, querying, and updating context data, and subscribing to changes in real-time.

#### 2.3.1.3 Data Models

FIWARE doesn't impose a fixed data model. Instead, it promotes the use of standards and open models, allowing developers to represent their data in a flexible and interoperable way. Data is represented as a set of **entities** with associated attributes and relationships, allowing the modeling of the actual world components, and relations among them.

#### 2.3.1.4 Pub/Sub Mechanism

FIWARE utilizes a **Pub/Sub** model for real-time data distribution (Figure 2.6). Components can subscribe to specific data updates, and the Orion Context Broker notifies them whenever changes occur.

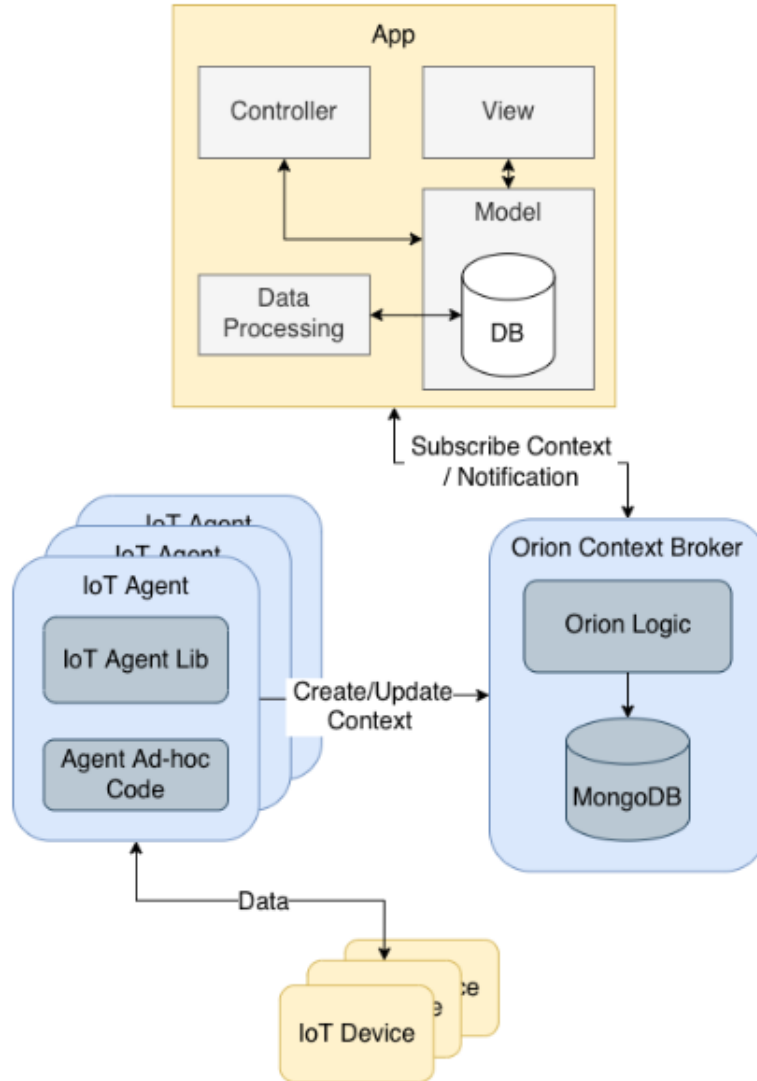


Figure 2.6: Simplistic overview of FIWARE's architecture

#### 2.3.2 BFT-SMaRt

BFT-SMaRt is an open-source Java library that implements **BFT** using **SMR**. It implements a modular design that separates consensus mechanisms, **SMR**, and other functionalities, thereby improving on maintainability and extensibility [16]. It also provides an **API** that can be used to integrate application logic into **BFT** model. As described by Bessani et al. [16], the objective with this project is to have a robust, easy-to-integrate,

and configurable solution that targets high-performance capabilities of modern hardware in fault-free execution environments while also ensuring correctness if a bounded number of replicas exhibit arbitrary behavior [16].

One key component of this library is the *Mod-SMaRt* module. It establishes a three-phase consensus logic similar to PBFT[6]. Clients send requests to all replicas and wait for their responses. This triggers the consensus leader to propose a message by multicasting it to the replicas(**PROPOSE**), which then triggers replicas to multicast hashes of the proposed requests in two phases - **WRITE**, **ACCEPT**[16]. The **WRITE** phase will serve as an acknowledgment from each replica that they see and validate the proposal sent by the leader node, while the **ACCEPT** phase serves to acknowledge that each replica received previous acknowledgments from other replicas, therefore fulfilling the agreement and allowing replicas to process the request(Figure 2.7).

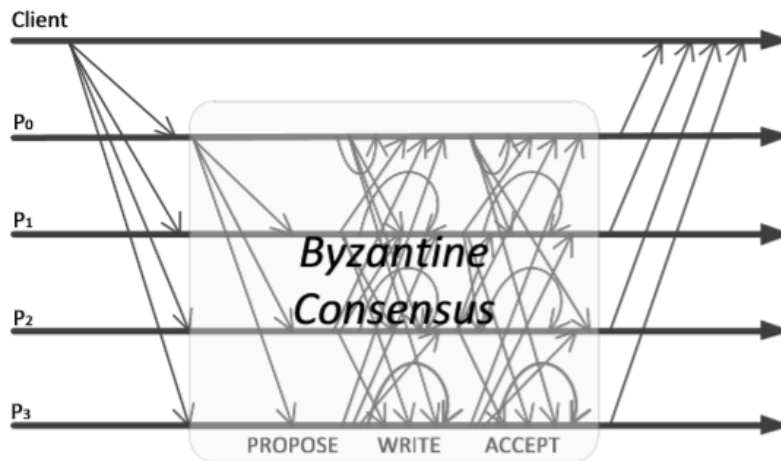


Figure 2.7: BFT-SMaRt's normal message exchange pattern[16]

## 2.4 Summary

This chapter describes the building blocks necessary to comprehend the scope of this dissertation by delving into topics such as IoT, fault tolerance, and generally describing the frameworks that will be used to develop the proposed solution. As such, it offers a broad contextual foundation for the subjects discussed in subsequent sections.

## Chapter 3

# State of the Art

In this chapter, work related to the integration of Byzantine fault tolerance in **IoT** systems is reviewed by providing a comprehensive description of current solutions. The discussion extends its scope to cover challenges posed by Byzantine actors on the network and also the methodologies employed to achieve fault-tolerant sensor readings. These aspects are considered crucial to understand the resilience and reliability requirements of **IoT** infrastructures in the presence of arbitrary faults.

### 3.1 Fault-Tolerant Sensor Readings

When building fault-tolerant **IoT** sensor systems, one must always assume sensors may fail either by completely shutting down or by arbitrarily providing erroneous readings. In classical Byzantine Fault Tolerance models, a client may try to confuse the system by sending conflicting messages[6, 16]; therefore, techniques must be employed to mitigate this risk. The following subsections outline key challenges and steps made to achieve fault-tolerant sensor readings and highlight the progress made so far in this research field.

#### 3.1.1 Sensor Fusion Problem

The Sensor Fusion problem is based on the idea of combining multiple sensor measurements to produce a single reliable estimate. Each sensor provides its reading (a measurement, an interval, a probability) from sensing the same physical variable, and the end-goal is to fuse data so that the final estimate has lower uncertainty than any individual sensor - better accuracy [23].

In a fault-tolerant context, some sensors may behave arbitrarily, potentially compromising the whole system, so a fusion algorithm must be capable of tolerating outliers. Moreover, even non-faulty sensors can exhibit slight variations in their readings due to inconsistencies related, for example, to accuracy capabilities and/or measurement timings. These discrepancies arise from factors such as manufacturing tolerances, environmental conditions, and inherent sensor limitations. As a result, sensors may produce readings that deviate slightly from each other, not because of faults, but due to these subtle differences. Many solutions to this problem view each sensor's output as an interval with bounds known to contain the true value (within the sensor's noise bounds). Then a central system computes an output (point estimate or interval) that is guaranteed to contain the true value as long as a bounded number of sensors are faulty[23, 24].

### 3.1.1.1 Marzullo's algorithm

Marzullo explores the idea of having sensors send intervals instead of single values in [24]. Sensors should output interval bounds from the current reading, taking into account inaccuracy. A system that receives this information evaluates received intervals and calculates the **fusion/intersection interval**: the leftmost and rightmost bounds that contain interval values from at least  $n - f$  sensors, respectively, thereby guaranteeing that the true value is contained in it. As shown in Figure 3.1, a higher number of faults negatively impacts the accuracy of the fusion interval, reflected as an increase in the distance between bounds.

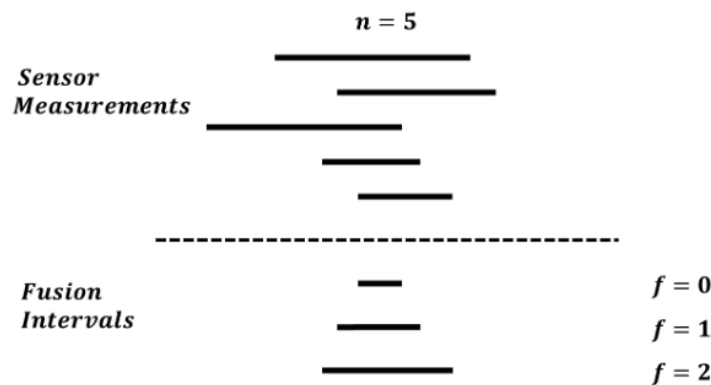


Figure 3.1: Marzullo's fusion interval according to the number of tolerated faults [25]

Marzullo proves that in systems with a majority of faulty replicas, the intersection interval can be more inaccurate than any of the sensors(Figure 3.2). However, if sensors

exhibit bounded inaccuracy, fault-tolerance against up to  $f$  sensors that behave arbitrarily (Byzantine) is achievable with a simple majority of sensors,  $2f + 1$ , with  $f$  being faulty. Although, as explained in [26], this comes with a cost: while higher replication, e.g.,  $3f + 1$ , allows the intersection of intervals to be more accurate than the most accurate sensor in the system,  $2f + 1$  prevents it from being higher than the most accurate of sensors. The paper shows that for  $2f + 1$  replication, the width of the interval will be at most the  $(f + 1)$ -th smallest of all sensors, while  $3f + 1$  replication extends this to only the correct sensors. Moreover, it is also proven that the reduced level of replication results in less accurate intervals than the most accurate sensor [25].

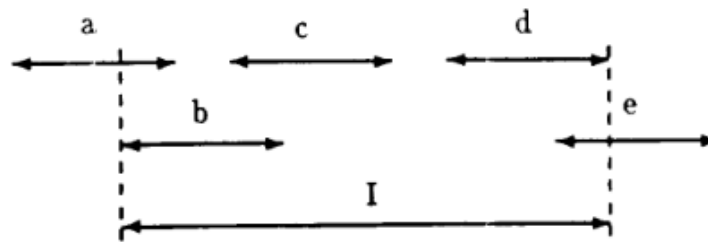


Figure 3.2: Marzullo's fusion interval inaccuracy problem if the majority of sensors are faulty [26]

In this paper, Marzullo addresses the sensor fusion problem by proposing its Fault-Tolerant Sensor Averaging algorithm. It also includes a comprehensive analysis of accuracy across different levels of replication, considering both bounded and unbounded accuracy models.

### 3.1.2 Approximate Agreement

The approximate agreement problem involves processes reaching outputs that are arbitrarily close to each other (within a bound) despite starting with different inputs and facing Byzantine faults. It relaxes exact consensus by ensuring agreement on values that fall within the convex hull of non-faulty initial inputs (validity). Typically, these protocols achieve convergence through iterative averaging or median selection [27].

#### 3.1.2.1 Approximate Byzantine Agreement (ABA)

Dolev et al. [27] address the challenge of achieving consensus in distributed systems where some nodes may behave arbitrarily. Unlike traditional Byzantine Agreement, this algorithm allows processes to decide upon values that are close to each other, given a specified acceptable deviation. Each sensor collects values from all other nodes and

stores them in an ordered set. To remove potential outliers, the  $f$  highest and lowest values are removed. Finally, the mean of these values is calculated by using a sample of the set.

```

Round 0 (Initialization Round):
  Input  $v$ ;
   $V \leftarrow \text{AsynchExchange}(v, 0)$ ;
   $v \leftarrow \text{mean}(\text{reduce}^{2t}(V))$ ;
   $H \leftarrow \lceil \log_c(\delta(V)/\epsilon) \rceil$ , where  $c = c(n - 3t, 2t)$ .

Round  $h$  ( $1 \leq h \leq H$ ) (Approximation Rounds):
   $V \leftarrow \text{AsynchExchange}(v, h)$ ;
   $v \leftarrow f_{2,t}(V)$ .

Round  $H + 1$  (Termination Round):
  Broadcast( $\langle v, \text{halted} \rangle$ );
  Output  $v$ .

Subroutine  $\text{AsynchExchange}(v, h)$ :
  Broadcast( $\langle v, h \rangle$ )
  Collect  $n - t$  round  $h$  responses:
    • Fill in values for halted processes.
    • Do not fill in default values.
  Return the multiset of responses.
  
```

Figure 3.3: Asynchronous approximation algorithm.[27]

As shown in Figure 3.3, the algorithm is processed in a multi-round way. Initially, the number of rounds  $H$  is calculated to determine how many are required, given an acceptable highest deviation bound. Subsequently, the previously described three-step process is repeated  $H$  times, progressively reducing the difference between obtained values. When this process ends, each node outputs its value. As proved by Dolev et al. [27], this algorithm solves the Approximate agreement problem, requiring  $3f + 1$  replicated nodes to achieve consensus.

### 3.1.3 Inexact Agreement

Inexact Agreement protocols further relax the Approximate Agreement validity condition by allowing the system to converge to values within a precision bound that might deviate from the initial inputs [28]. Similarly, they also employ iterative averaging or median-based filtering to improve precision [28].

#### 3.1.3.1 Fast Convergence Algorithm (FCA)

FCA is presented by Mahaney and Schneider[28] as a solution to the Inexact Agreement problem. It operates by identifying a set of acceptable values from received measurements. It then replaces values deemed unacceptable with a computed value (such as the

average, median, or midpoint) derived from these acceptable values and finally returns the mean of this last multiset(Figure 3.4). This approach aims to represent data from correct processors while minimizing the impact of incorrect sources, thereby focusing on enhancing precision in the distributed system [23, 28]. Graceful degradation is also a relevant aspect of this protocol. In scenarios where there are between one to two-thirds faulty nodes, it is capable of detecting, reporting, or continuing to compute values that adhere to accuracy and precision constraints [28].

**Fast Convergence Algorithm:**

1. **Request and receive a value  $v_{rp}$  from every other processor  $r$ .**  
Let  $V_p$  be the multiset of these values.
2. **Construct the acceptable set  $A_p$  from  $V_p$ .**
3. **Compute  $e(A_p)$ .**
4. **Replace any  $v_{rp}$  in  $V_p$  that is not in  $A_p$  by  $e(A_p)$ .**
5. **Set  $v_p'$  to the average of the values in  $V_p$ .**

Figure 3.4: Fast Convergence algorithm - process execution.[28]

As shown in the paper[28], it requires  $3f + 1$  nodes to participate in the protocol given  $f$  faulty ones to guarantee inexact consensus. It is also concluded that compared to Dolev's ABA algorithm FCA provides better precision while also being considered more robust since it implements graceful degradation capabilities.

#### 3.1.4 Brooks–lyengar Algorithm

Marzullo's sensor fusion algorithm introduces the idea of using intervals instead of single point values and focuses on improving the accuracy of sensor readings [23, 24, 26]. ABA and FCA protocols are more concerned with enhancing the precision of sensors and guaranteeing the reliability of the whole system. The emphasis observed in later algorithms is justified in contexts where ensuring processor coordination and system consistency\* is considered more critical than achieving higher levels of value accuracy[29].

In [29], Brooks and lyengar describe both concepts of Precision and Accuracy by making an analogy to dart shots. As shown in Figure 3.5, the dartboard on the left represents Accuracy, depicted as the distance between each shot to the center of the board. The

---

\*Consistency can be regarded as the concept of Interactive Consistency explained in the Byzantine Generals Problem described in the Background section

right one highlights the area where all the darts are contained (convex hull). In the context of sensor networks, accuracy refers to the deviation of a sensor's reading compared to the true physical real-world value, whereas precision denotes the degree of deviation between all sensor values[23, 26, 28, 29].

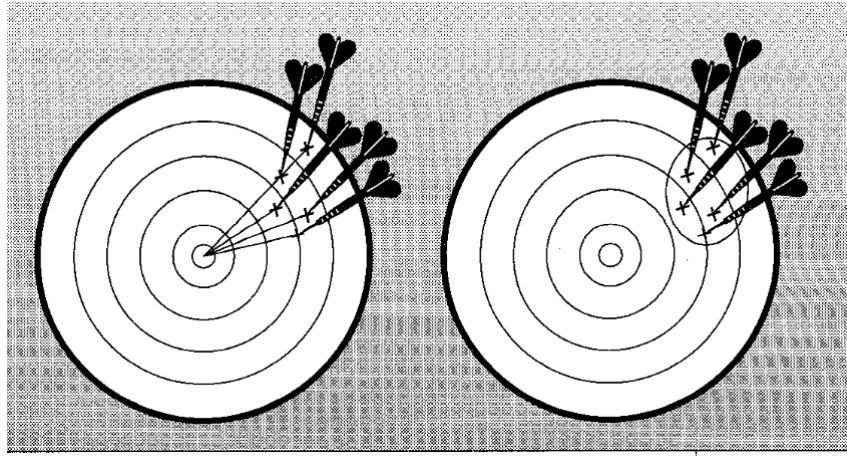


Figure 3.5: Dart short analogy to explain Precision versus Accuracy concepts.[29]

The Brooks-Iyengar algorithm implements methodologies used in both Sensor Fusion, Approximate Agreement, and Inexact Agreement algorithms - thereby being described by the authors as a hybrid algorithm [29]. Initially, each sensor acquires its measurement and generates an interval based on its inaccuracy bounds. All sensors then exchange these intervals to all other peers so that each node can compute the intersections between intervals. For a given interval, if it intersects with fewer than  $N-f-1$  others, it is considered faulty and discarded from the set of potentially acceptable. This guarantees that there is at least a simple majority consensus for a given interval. After isolating faults, the resulting data is aggregated using a weighted average of each interval's midpoint: the sum of all midpoint values, each multiplied by the number of sensors that overlap it, divided by the number of midpoints. If necessary, this algorithm can be iterated to adhere to acceptable precision bounds.

Through the combination of methodologies described in previous algorithms, Brooks-Iyengar is capable of achieving both higher precision and accuracy while also providing the same level of robustness described in FCA through graceful degradation [23, 29].

## 3.2 Byzantine Fault Tolerance in Publish/Subscribe systems

Implementing Byzantine Fault Tolerance in Pub/Sub systems is a non-trivial but desired approach since it can be adopted for many applications[30]. This section reviews some

implementations, starting with the integration of **BFT** in database systems, reviewing the Byzantium middleware. It then explores state-of-the-art **Pub/Sub** solutions focusing on aspects that are pertinent to the system proposed in this work.

### 3.2.1 Efficient Middleware for Byzantine Fault Tolerant Database Replication [31] - Byzantium

Garcia et al. describe Byzantium as a middleware that can be integrated with database systems to incorporate **BFT**, ensuring resilience against arbitrary faults. It assumes an asynchronous environment where messages can be delayed, corrupted, or lost, but with guarantees of safety properties and eventual liveness for periods of bounded delay. The paper first describes its database model regarding isolation levels. Specifically on the concept of Snapshot Isolation (**SI**), introduced by Berenson et al. [32], where concurrent threads operate on a snapshot of the last committed state and only the first thread ready to commit succeeds, while others abort. This allows for improvements in performance since threads can execute concurrently until a commit is performed in the database. Also, read operations don't abort or need to acquire locks provided their snapshot is consistent throughout the operation execution[31, 32].

As shown in Figure 3.6, the system follows traditional **BFT** protocols [6, 16] with  $3f + 1$ -replication with at most  $f$  faulty components. Bizantium Client Proxy instances provide a seamless interface for clients to communicate with the database system and multicast calls to Byzantine replicas. It encapsulates a **BFT** client and uses its underlying interface to forward requests to **BFT** replicas to trigger the **BFT** protocol. Bizantium Replica Proxy instances are used to run these replicas and the **BFT** protocol to achieve consensus and agree upon an ordered set of database operations. Provided consensus is reached, each non-faulty replica communicates with its database to perform the agreed database call, ensuring state consistency across all databases.

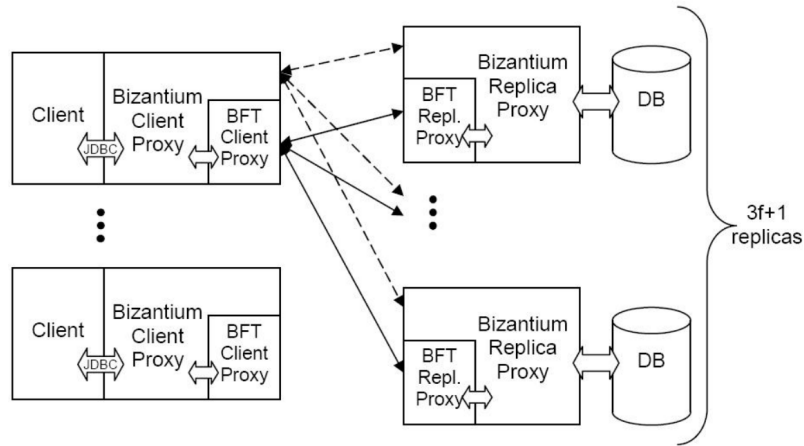


Figure 3.6: Byzantium system architecture.[31]

### 3.2.2 Byzantine Fault-Tolerant Publish/Subscribe: A Cloud Computing Infrastructure

Tiancheng Chang and Hein Meling[30] make a comprehensive evaluation of failure scenarios that should be considered when developing Byzantine Fault Tolerance in Publish/Subscribe (**BFT-PB**) systems. The authors start by describing various threats posed by Byzantine Brokers, potentially disrupting system operations. Such brokers may delay messages indefinitely, degrading system performance or even leading to complete system failure. Furthermore, they can reorder publications and subscriptions, thereby compromising system consistency. In addition, brokers may corrupt subscriptions by tampering with their information, altering, dropping, or even adding erroneous ones. These actions might result in message misdelivery to unwanted subscribers or even threaten publication integrity, ultimately harming the trustworthiness/correctness of data. Byzantine Publishers' attacks are also referenced as potentially affecting system performance and integrity in several ways. One such described way is through the injection of spam, where a malicious publisher pushes irrelevant information to brokers in the hope of increasing resource consumption and degrading system performance. This can also be amplified if it can collude with malicious brokers, therefore bypassing, for example, filtering validations. On the other hand, a similar publication flooding attack is presented where publications that match a wide number of subscriptions are pushed to exhaust broker resources and possibly undermine the system's **QoS**. Finally, the authors reference an attack where faulty publishers collude with faulty subscribers to generate huge amounts of publications. These are already subscribed to degrade system performance and potentially cause a denial-of-service. Moving on to Byzantine Subscribers, these can attack

the system's responsiveness simply by subscribing to all types of publications (greedy subscription) or by repeatedly subscribing and unsubscribing (frivolous subscription) to publications. Another possibility is to deny receiving publications, thereby triggering a retransmission and increasing broker load. While the authors argue that some of these attacks are difficult or even impossible to correctly avoid, replication can be used to mitigate some of them. By using **SMR** techniques in conjunction with replicated brokers, the authors argue it is possible to prevent single points of failure as long as a majority of brokers stay responsive and consistent. However, it is also pointed out that the use of replication poses some challenges regarding scalability, and because of the challenging scenario of adapting **BFT** protocols that require time and synchronization coupling to **Pub/Sub** event-based architectures, which are mostly asynchronous.

### 3.2.3 Trinity [33]

The Trinity system is presented by Gowri Sankar Ramachandran et al. [33] as an enhancement to the existing publish/subscribe models, particularly by adding byzantine fault tolerance and data immutability in multi-stakeholder environments. The author highlights the motivation for developing systems with these properties by arguing that typical publish/subscribe applications often rely on centralized brokers, which inherently create a single point of failure. This architecture can increase the system's vulnerability to targeted attacks, therefore emphasizing the need for more resilient and decentralized alternatives.

Trinity distributes the system of replicated brokers into domains comprised of multiple components. Apart from the broker, each domain consists of a Consensus node that belongs to the blockchain network and is used to achieve consensus and to leverage the distributed ledger for immutable storage, and a collection of publish/subscribe clients that communicate within their local domain through the broker (Figure 3.7).

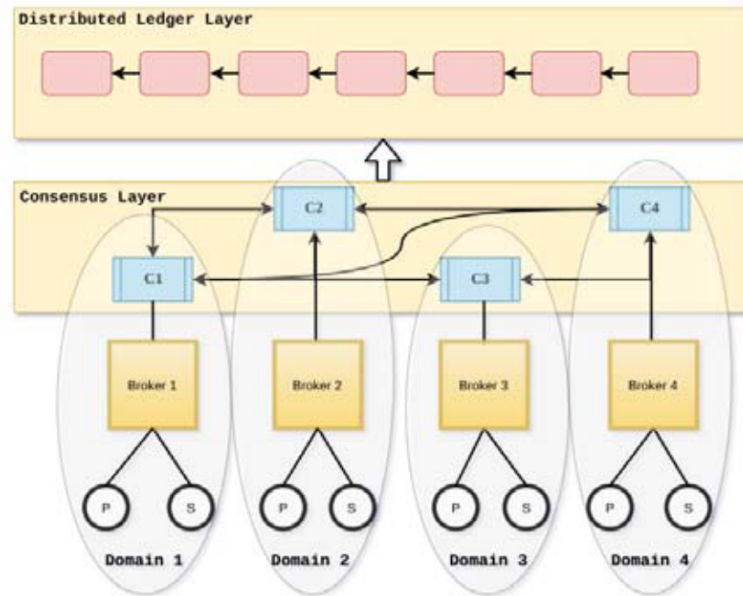


Figure 3.7: Trinity system architecture. [33]

An aspect mentioned in the paper [33] emphasizes that the proper functioning of the architecture depends on the absence of faults in either components within a domain. Since it is also assumed that the system follows traditional **BFT**, fewer than one-third of the domains can be faulty: this means the system cannot withstand any two or more faulty components at the same time in different domains.

Although it is shown in the paper that there are costs associated with adding a consensus layer, the author acknowledges that the benefits of introducing fault-tolerance and immutability capabilities can outweigh these. Not only that, it is also mentioned Trinity's capability of being blockchain-agnostic, therefore allowing for a more modular "plug-and-play" architecture.

### 3.2.4 Galaxy [34]

Galaxy is presented by Zhang et al. [34], as a solution to tackle byzantine faults and data privacy requirements in **Pub/Sub IoT** systems. It is designed for multi-organization scenarios where participants provide infrastructure to the network while also ensuring data privacy constraints.

The system consists of a governance layer that makes use of a high-performance blockchain for cross-shard message total order, ensuring consistency in multi-shard operations. Additionally, it includes a secret key sharing service to enable privacy-preserving

mechanisms. As explained in the paper [34], this layer acts as a ledger responsible for global management and coordination of tasks in the system, thus requiring fewer nodes. Nevertheless, it still follows the **BFT** model where less than one-third of the network can become faulty.

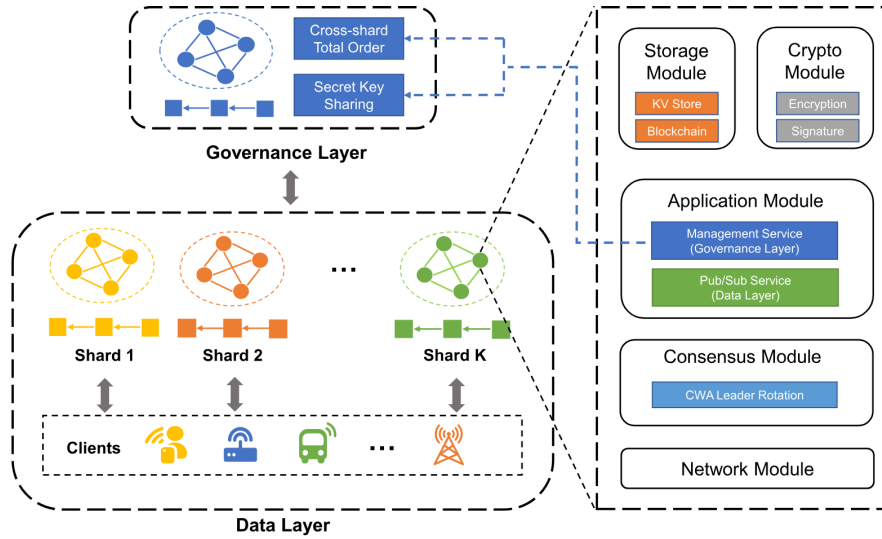


Figure 3.8: Galaxy's system architecture. [34]

Adding to this, there is a data layer responsible for handling the **Pub/Sub** service. As shown in Figure 3.8, the system is organized into shards, consisting of organization-provided nodes. In addition to the **Pub/Sub** application, each node implements a series of modular components used to provide storage, cryptography, consensus, and networking capabilities to the system participants. Distribution of nodes between shards is done through the use of a novel, partially random algorithm to avoid scenarios where one organization controls a majority of a shard's node count and can potentially compromise the protocol through collusion.

Another contribution provided by the work related to this paper is the improvement in the leader selection mechanism. The algorithm entitled Crash and Withholding Avoidance (**CWA**) provides a more effective solution to this scheme by better avoiding selecting failed/faulty nodes as leaders[34]. The faulty leader can affect the system's liveness by not responding since it is crashed or by purposely withholding from proposing messages. The author describes this as a crucial step to improve the system's overall performance, particularly because the **BFT** algorithm used in this system relies heavily on the leader rotation process. This motivation similarly applies to other leader-based **BFT** algorithms, as

they rely on timeout mechanisms to trigger leader re-elections, during which the system may experience unresponsiveness.

The solution is shown by the author to improve throughput and scalability through the combination of sharding, which enables parallel consensus execution, and the **CWA** algorithm, which avoids frequent faulty leader election.

### 3.3 Summary

This section presented a comprehensive overview of solutions to both the Sensor Fusion, Approximate and Inexact Agreement problems while also reviewing existing solutions that integrate **BFT** into **Pub/Sub** systems.

It began by reviewing Marzullo's work, which showed that for bounded inaccuracy scenarios, it is possible to solve the Sensor fusion problem while requiring only  $2f + 1$  sensors. Additionally, it introduced the idea of using intervals instead of concrete values. Dolev's **ABA** algorithm solves Approximate Agreement, focusing on precision. This is particularly relevant when processor coordination is deemed crucial to ensure system consistency. **FCA** improves Dolev's work by reducing the impact of faulty nodes through replacing unacceptable values with a computation of the possibly correct ones. It also introduces the concept of graceful degradation, where the system stays responsive when at most two-thirds of the network is faulty. Finally, the Brooks–lyengar algorithm is presented as a hybrid solution that combines methodologies from previous algorithms, both improving precision and accuracy.

Moving to **BFT Pub/Sub** systems, the Byzantium architecture is briefly overviewed, given its notable resemblance to part of the system presented in the work of this study. The remaining reviewed papers offer valuable insights into state-of-the-art implementations of Byzantine fault tolerance in publish/subscribe systems, reinforcing the motivation for pursuing such resilience. In addition, they provide a clear overview of the various assumptions underlying the design of these systems, offering a perspective on fault model considerations.

## Chapter 4

# Architecture

This chapter outlines the theoretical aspects of the proposed system by presenting concrete reasoning for deployed strategies and describing both the system and fault models. It identifies potential problems with naive solutions and presents a refined architecture that addresses these issues. The adopted framework is also reviewed to provide a clear context for system components.

### 4.1 IoT Architectures Under Faults

The traditional **Pub/Sub** architecture is examined here, specifically in relation to its limitations in tolerating faults. This provides the groundwork for designing and justifying our proposed architectural refinements.

#### 4.1.1 Non-replicated **Pub/Sub** Architectures

As specified previously, the **Pub/Sub** model is a widely used messaging pattern in distributed systems, particularly for asynchronous, decoupled communication in applications. In a typical **Pub/Sub** architecture, the system consists of three main components: publishers, brokers, and subscribers (Figure 4.1).

1. **Publishers**: Entities (e.g., sensors) that send messages (events or data) to the system without knowledge of who will consume - space decoupling[35].
2. **Message Broker**: Acting as the intermediary, the broker manages message routing. It receives publications, stores them temporarily if needed, and forwards copies to all

relevant subscribers. Brokers often support features like message queuing, filtering, and load balancing to handle scalability.

3. **Subscribers:** These are consumers (e.g., applications) that express interest in specific topics by subscribing to them. Subscribers receive messages asynchronously - synchronization decoupling[35] - from relevant topics, without direct interaction with publishers[35].

Information can be grouped into topic-based, content-based, type-based variants (e.g., temperature updates in an IoT system)[35]. The flow is decoupled: Publishers "publish" topics to the broker, brokers store those topics, and subscribers "subscribe" to them, receiving pushed notifications whenever a new change is made. This enables scalability and flexibility, as components can join/leave dynamically without disrupting others - time decoupling[35].

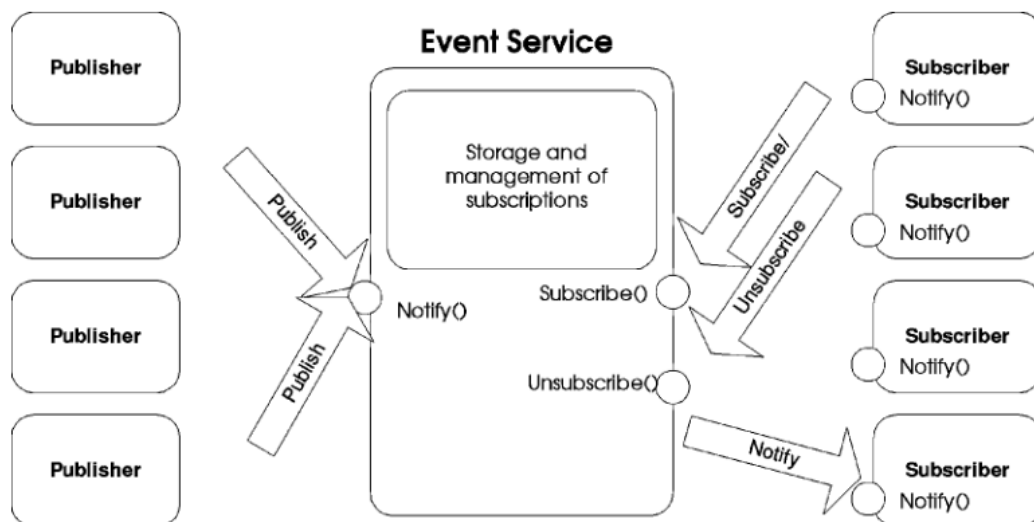


Figure 4.1: Simplified depiction of Pub/Sub systems' architecture[35]

#### 4.1.1.1 Fault Model

While Pub/Sub excels in decoupling and scalability, it is inherently limited in handling faults. A faulty broker can drop, duplicate, or alter messages without detection, potentially violating the system's liveness, safety, and correctness properties. The combination of these factors makes it impossible for consumers to distinguish valid from malicious data. A system with non-replicated brokers acts as a single point of failure, making it more

susceptible to being compromised. If Byzantine behavior is taken into consideration, attackers can perform replay attacks, tamper with message data, or selectively deliver to subscribers, breaking system integrity.

On the publisher side, these can propagate incorrect data and/or flood the system with excessive messages, thereby hindering the system's reliability. Since there is no replication or consensus to establish a quorum-based decision on messages, and there are no proofs that validate the correctness of this information, such attacks can not be detected by subscribers, which entirely undermines the system.

Therefore, traditional non-replicated publisher-broker-subscriber architectures are not designed to handle faults, whether benign or Byzantine.

#### 4.1.2 Introducing Broker Replication

One way of improving fault tolerance is to have multiple broker instances. Replication involves duplicating message data and distributing broker responsibilities across multiple nodes, ensuring redundancy and message durability, therefore improving system resilience.

One example of this architecture, Kafka [36], divides each topic into partitions, which are distributed across a configurable number of replicated brokers. Each partition has one leader broker that handles all reads and writes, while the remaining are followers, which replicate the leader's data to ensure fault tolerance. These replicas are kept in sync with the leader so that if it fails, one of them can take over - Figure 4.2.

Publishers and subscribers in Kafka communicate directly with brokers: producers send messages to the partition leader, while consumers receive messages from that same leader and remaining follower brokers. Replication does not alter the interaction model, but adds a layer of fault tolerance: followers maintain synchronized copies of the partition so that, if a leader fails, another broker can transparently take over. From the client's perspective, communication remains identical to that of non-replicated architectures, as replication is managed internally by the brokers. However, consumers must specify the required level of acknowledgment, i.e., how many broker replicas must confirm an update before it is considered valid.

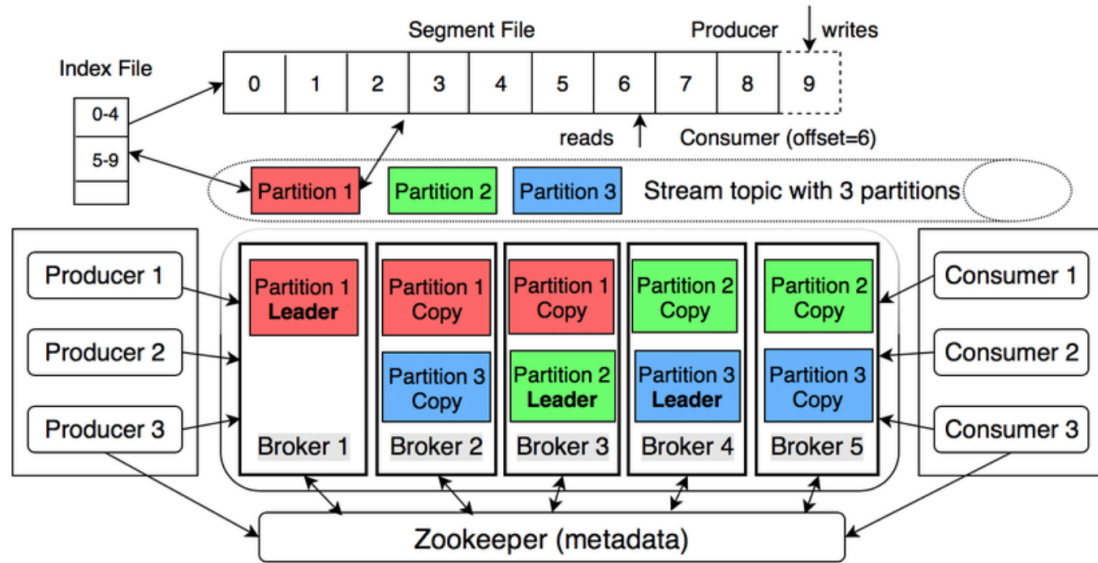


Figure 4.2: Kafka architecture[37]

#### 4.1.2.1 Fault Model

Compared to non-replicated architectures, this primarily addresses crash-fault scenarios where brokers fail by halting without malicious intent. It still falls short against Byzantine faults as it lacks built-in mechanisms to detect or mitigate active malicious behavior, such as:

- **Malicious Brokers:** A Byzantine broker could drop messages selectively, replay old ones, or inject false data without detection, as replication focuses on data duplication without cryptographic verifications. For example, if an attacker controls a partition broker leader, it might delay acknowledgments or corrupt replicas, violating consistency without triggering failover.
- **Coordinated Attacks:** Broker replication does not prevent collusion among faulty brokers, which could lead to quorum-based attacks where multiple brokers forward different/wrong information to confuse consumers.
- **Faulty Publishers:** Publishers might crash, leaving the system without any updates regarding some context entity, or even, if byzantine, they could intentionally omit or publish wrong values to the broker system.

To mitigate a scenario where a malicious leader broker does not share received information with replicas and actively refrains from notifying subscribers, one could naively

adapt the system to force publishers to forward messages to all broker replicas. While this approach could resolve the presented issue, it introduces problems related to the consistency of the replicated state. Under the asynchronous and decoupled **Pub/Sub** model, message transmission delays can cause brokers to receive events in different orders at different times. As a consequence, local states across brokers may diverge, leading to inconsistent information being provided to subscribers, which undermines the correctness of the system.

While this architectural model tolerates crash faults in the broker layer, it does not address non-benign behavior from publishers or brokers. As publishers are not replicated, they may inject misleading data or withhold information entirely without being detected. As such, broker replication alone is insufficient to protect against Byzantine behavior across the system, leaving publisher faults unmitigated.

## 4.2 Naive Approach

As presented previously, non-benign faults cannot be resolved solely through replication of brokers, and more specifically in leader-based replication scenarios, as brokers can still act as single points of failure or deceive non-faulty follower replicas. Assuming that components can behave maliciously, the following architecture extends replication across all components. Specifically, sensors and brokers also need to be replicated and operate in parallel without requiring intra-layer coordination (contrary to leader-based architectures): sensor readings are forwarded independently, and brokers maintain a state for the corresponding context data.

### 4.2.1 Architecture Concept

The architecture follows the Byzantium[31] strategy by incorporating Byzantine fault-tolerant components between the individual **Pub/Sub** layers. Its overall structure follows the topology illustrated in Figure 4.3.

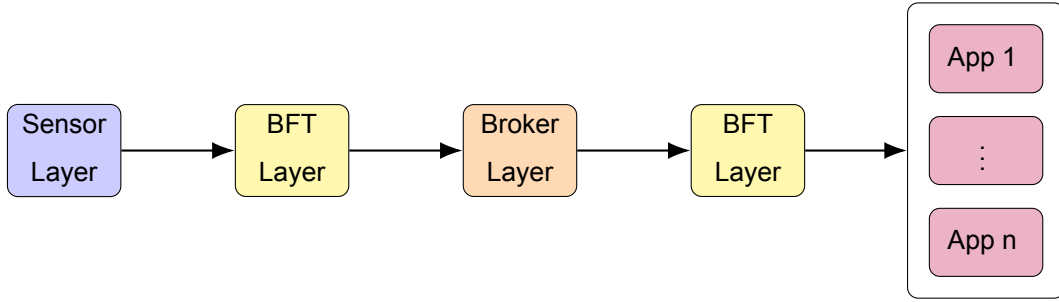


Figure 4.3: Naive architecture: Sensors transmit data to the **BFT** layer, where consensus is executed to establish a consistent ordering of requests. Once agreed upon, the data is delivered to the broker layer. Notifications are then generated for subscribed applications, but are first routed through a second **BFT** layer, which verifies the consistency of the broker's state. After this additional consensus round, applications receive the notifications and confirm correctness by validating that a quorum of matching values was obtained.

A major drawback of naively replicating components is that requests may arrive at different times, potentially causing state inconsistencies. To address this, **BFT** consensus layers are placed between sensor-broker and broker-application layers, providing total message ordering, promoting consistent state replication across brokers. Compared to other architectures, such as Trinity[33], which implements consensus logic directly within the broker instance, this approach decouples both while still ensuring consistency provided by the consensus mechanisms. This promotes a more modular and agnostic implementation, which prevents coupling to specific technologies or frameworks.

#### 4.2.1.1 System flow

Sensors send their readings to the **BFT** layer, which is responsible for initiating consensus and establishing a total order of requests. The agreed results are then disseminated to all brokers, ensuring that their state remains consistent 4.4. Since each layer follows traditional **BFT** replication with  $3f + 1$  replicas, an aggregation mechanism is required to ensure that at least a quorum of  $2f + 1$  sensors have sent matching values. Importantly, this fault model assumes that components from each layer may exhibit Byzantine (malicious) behavior, and the replication and consensus protocols are used to mitigate those faults.

Upon updating context information within a broker, the value is propagated to subscribed applications. However, this propagation first passes through an additional **BFT** layer to ensure agreement on broker values: the prior consensus process is repeated until replicas attain majority agreement, after which the final response is disseminated.

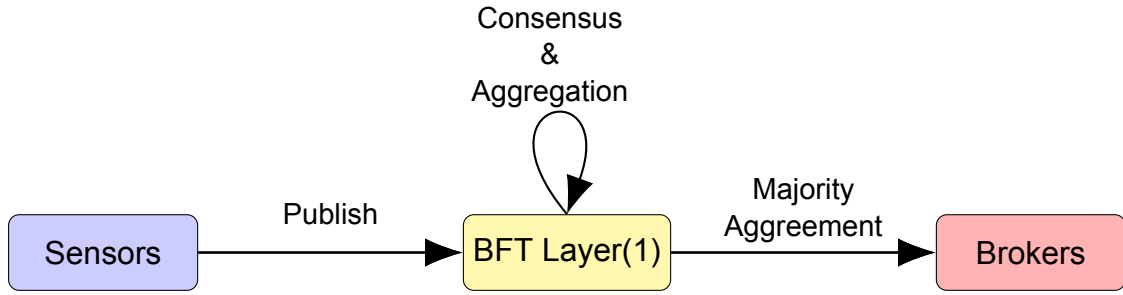


Figure 4.4: Message model for Entity Creation or Update: When sensors create or update entities, a request is proxied to a BFT layer to ensure value consistency all across the broker layer.

Applications can verify the validity of responses by confirming receipt of at least  $f + 1$  matching values 4.5.

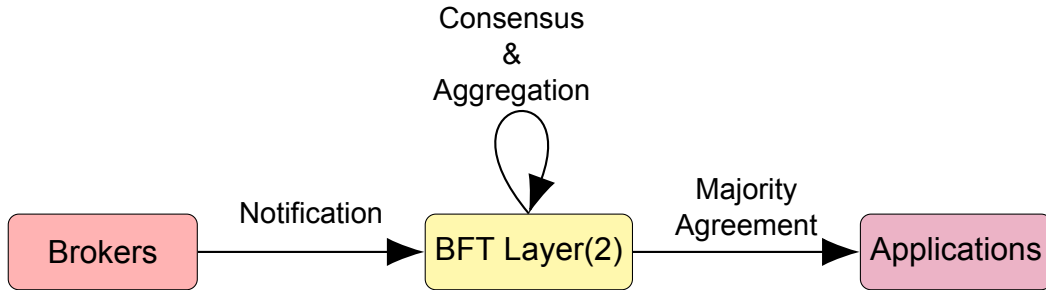


Figure 4.5: Notifications Message Model: Upon receiving a context update, the broker layer exchanges information with the subscribed application; however, to ensure consistency, values are proxied through another BFT layer.

Applications interact with the system by subscribing to topics or by querying its information. Such requests must be multicasted to the second BFT layer to ensure malicious faults are covered. Upon request, applications receive responses directly, and must obtain  $f + 1$  matching responses to ensure they are consistent 4.6.

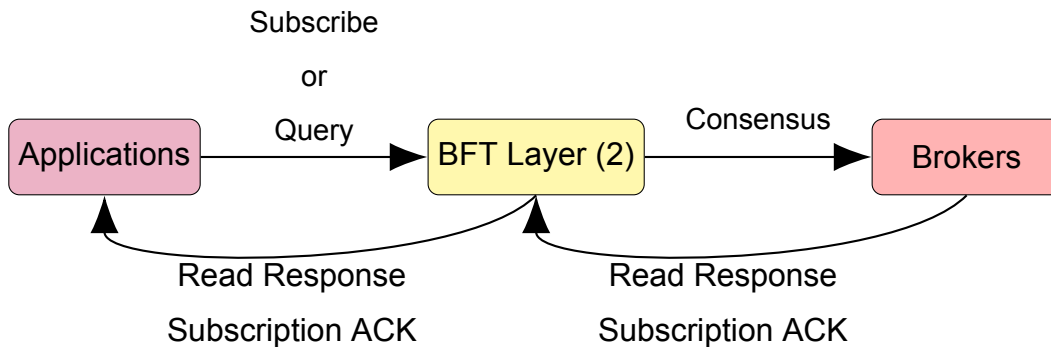


Figure 4.6: Application Subscription and Query Message model: Applications can subscribe to given topics through the second BFT layers. Both context query responses and subscription acknowledgments are directly forwarded to applications. However, upon subscription, notification requests are always handled by the second BFT layer, following the consensus and aggregation process described previously.

#### 4.2.2 Middleware-Based Architecture

Another key objective of this design is to deliver a middleware-based solution that abstracts consensus-related mechanisms entirely from both publishers and subscribers, ensuring seamless integration and transparency. As such, sensors must not be required to implement any consensus-specific logic, and applications are only required to decide upon a quorum of received notifications. Sensor logic operates unaware of whether it is publishing directly to a broker: sensors issue requests using the same interaction patterns provided by the underlying broker API. This approach intentionally delivers a transparent and agnostic solution to possibly accommodate other Pub/Sub frameworks and/or consensus libraries (Figure 4.7).

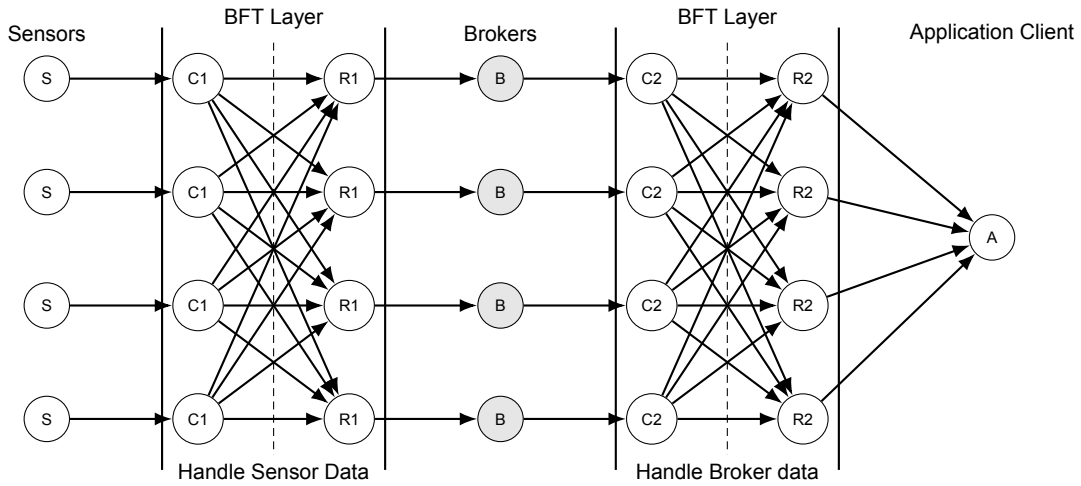


Figure 4.7: Middleware-based Architecture model: All components are replicated in a  $3f + 1$  manner apart from applications. Sensors(S) send sensory data to BFT Layer clients(C1) in 1-to-1 communication. These will broadcast requests to BFT Layer Replicas(R1) that run the consensus and aggregate agreed-upon values until a  $2f + 1$  threshold is met, after which they are propagated to Brokers in 1-to-1 communication. The same process is repeated in the second BFT Layer for Broker values. This latter is responsible for validating the consensus value and notifying Applications that have subscribed to a given topic. Also, it provides the interface for applications to subscribe to topics.

As shown in Figure 4.7, to support this concept, two components are introduced. The *BFT\_Broker\_Proxy* is responsible for executing the consensus protocol and forwarding aggregated consensus data to the broker layer. While the *BFT\_Broker\_Client*, acts as a middleware interface that mimics the broker API and is responsible for invoking consensus upon received requests. The latter abstracts consensus mechanisms, allowing applications to interact with the system as if they were communicating directly with a broker.

#### 4.2.2.1 BFT\_Broker\_Proxy

The *BFT\_Broker\_Proxy* acts as a Byzantine fault-tolerant server replica that leverages the BFT-SMaRt[16] library to provide consensus-based state machine replication for FI-WARE Context Broker operations. This component receives context operations (create, update, query, subscribe) from **BFT** clients, processes them through the Byzantine consensus protocol providing total ordering in requests, and upon achieving agreement over a quorum of received values, forwards the validated operations to the underlying context broker. It implements operation aggregation mechanisms through the *OperationSet* class, to validate if there was a quorum of  $2f + 1$  different **BFT** client invocations that have seen a matching value reading from each corresponding sensor invocation. This condition is essential for preventing malicious publishers from overwriting the broker state with conflicting messages, which could corrupt the sensory data propagated through the system.

Another important role of these proxies is to ensure consistency across broker notifications sent through **BFT** clients. Similarly to previous context operations, notifications require a quorum of  $2f + 1$  values to represent that a majority of sensors have sent a consistent reading which was successfully propagated to each broker. Therefore, the replicas will execute the same post-consensus aggregation process until the specified threshold is reached. If so, each replica forwards the aggregated value to subscribed applications, which accept it if there is a valid quorum of matching responses.

The reasoning behind the given threshold( $2f + 1$ ) follows traditional **BFT** requirements: if any application receives a  $2f + 1$ -sized quorum of values in the worst case scenario,  $f$  of those are wrong; as there are still  $2f + 1 - f = f + 1$  valid messages, it ensures there is a majority of values that outweigh the malicious ones:

$$\forall f \geq 0, \quad f + 1 > f$$

As such, applications can decide which value to accept. If we assume a smaller threshold, we see this condition doesn't hold: if a  $f + 1$ -sized quorum of messages is received and  $f$  are malicious, the receiver will have  $f + 1 - f = 1$  valid values. For any  $f > 0$ , the number of correct responses is smaller than or equal to the wrong ones. Therefore, either the application won't be able to decide upon which value to accept( $f = 1$ ), or it might take a wrong value since it sees the majority of them( $f > 1$ ).

#### 4.2.2.2 BFT\_Broker\_Client

The *BFT\_Broker\_Client* serves as a middleware proxy that mimics context broker interfaces. This component operates as an intermediary layer that intercepts FIWARE NGSI-V2 API calls from IoT applications and triggers consensus by incorporating BFT-SMaRt[16] library methods. Additionally, it acts as a subscription handler to inform brokers of the intent to subscribe to a topic and to allow notifications to reach the desired subscriber(s).

Similar to the *BFT\_Broker\_Proxy*, clients have separated roles for both applications and IoT(sensor) layers. While in the first, they act as subscription maintainers - keeping a subscription mapping to handle notification requests to each subscriber; in the second, the responsibility is in providing an interface to propagate sensor data through consensus invocation without requiring sensors to implement this logic. This becomes relevant not only because it decouples consensus from application logic, providing a more modular architecture, but also aligns with the inherent hardware-constrained environment of IoT systems.

#### 4.2.3 Fault Model

Although this architecture can provide higher fault tolerance, as expected, not only does the use of multiple layers hinder scalability, but it also fails to address a more capable attacker model where nodes can collude to undermine the system, potentially disrupting it. Additionally, the replication of sensors introduces potential problems regarding value alignment due to the inherent physical properties of these components.

##### 4.2.3.1 Sensor Inherent Inconsistencies

The replication of sensors inherently introduces value inconsistencies, as measurements of the same physical phenomenon are rarely identical due to noise, calibration differences, or environmental disturbances[24]. These discrepancies do not necessarily indicate faults but rather reflect the natural imprecision of sensing devices. To mitigate this, this architecture model assumes a predefined precision-based tolerance range within which sensor readings are considered valid and equivalent. Values falling outside of this range are classified as faulty and treated as potential Byzantine values. This approach enables the system to distinguish between benign imprecision and actual sensor faults, ensuring that

replication enhances reliability without being compromised by unavoidable measurement variations.

#### 4.2.3.2 Network Complexity

The use of two **BFT** layers not only increases hardware requirements, but also significantly raises bandwidth demands. As demonstrated in Table 4.1, analyzing the network complexity in the worst-case scenario, considering  $n$  nodes for each layer, one single context create operation and broker notification to a single subscriber requires  $2n^2 + 4n$  messages. This scenario excludes the network messages associated with the consensus protocol, as it may vary depending on the specific **BFT** protocol employed; however, the number of consensus executions can severely impact system performance, as it requires multiple rounds of inter-replica communication to exchange information.

To reduce hardware operational costs, the system could use a single **BFT** layer for both execution parts; however, this approach would not lower network complexity since the same actions are performed. Furthermore, it could adversely affect overall system performance, as this centralized consensus layer could become overloaded with requests from both sensor readings and broker notifications, thereby throttling operations and acting as a bottleneck to the whole system. This could not only make the system unreliable but also hinder scalability.

| Layer        | # Nodes ( $n$ ) | Direction of Communication | # Messages                    |
|--------------|-----------------|----------------------------|-------------------------------|
| Sensor Layer | $n$             | To all C1 nodes            | $n$                           |
| BFT C1 Layer | $n$             | To all R1 nodes            | $n^2$                         |
| BFT R1 Layer | $n$             | To all B nodes             | $n$                           |
| Broker Layer | $n$             | To all C2 nodes            | $n$                           |
| BFT C2 Layer | $n$             | To all R2 nodes            | $n^2$                         |
| BFT R2 Layer | $n$             | To Application Client      | $n$                           |
| <b>Total</b> |                 |                            | <b><math>2n^2 + 4n</math></b> |

Table 4.1: Worst-case network message complexity per round from sensor to application client using the architecture depicted in Figure 4.3.

Another aspect is the redundant execution of a second consensus for broker notifications. Provided a consensus was already reached, running another one seems like a costly solution; however, since there is no guarantee that brokers won't act maliciously, a

quorum of seen values is again required to be agreed upon. Nevertheless, as elaborated in the following section, this solution has the unintended consequence of increasing the system's proneness to faults.

#### 4.2.3.3 Fault propagation

When building Byzantine proof multi-layer replication systems, it is not only expected that each layer might contain a faulty replica, but also that somehow an attacker can spread its control through various layers. While some designs assume no collusion between layers[38] or couple certain inter-layer faults[33], this architecture explicitly allows layers to operate independently and thus be subject to cross-layer compromise. As a result, an attacker can take control of replicas in distinct layers and coordinate them to hinder consensus, thereby threatening the system's Liveness, Safety, and Correctness properties.

As shown in Figure 4.8, the existing architecture is not capable of withstanding inter-layer fault scenarios. Even though each layer respects the Byzantine threshold of less than one-third faulty replicas, the propagation of wrong values can still hinder value correctness. Non-faulty nodes could be unaware of malicious behavior and disseminate wrong values, acting as unintended malicious nodes. This may not only happen for sensor values, but could also happen if a malicious broker and BFT client collude to propagate wrong values and undermine the consensus majority decision.

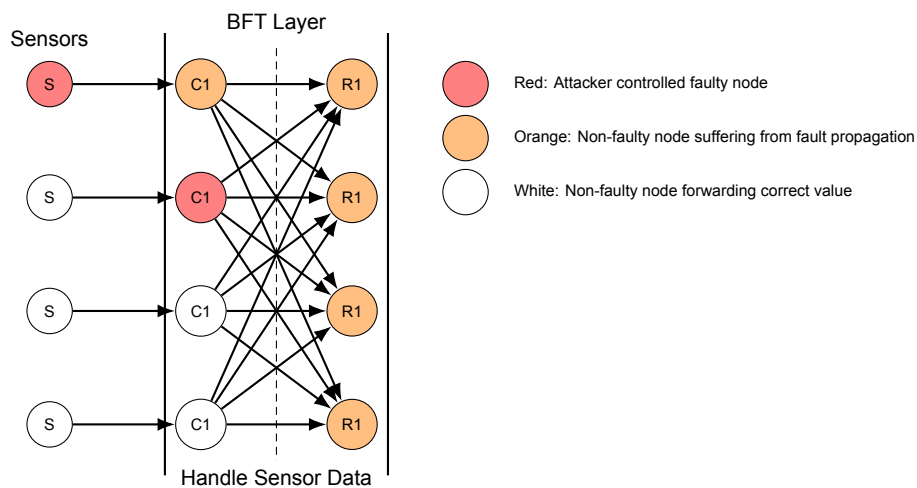


Figure 4.8: Fault propagation scenario 1: Attacker replicas collude to undermine the system's liveness. A malicious sensor purposely sends a wrong value to its BFT client, triggering a consensus execution. Furthermore, a malicious BFT client broadcasts the same wrong value to consensus replicas. Even if the remaining BFT clients are not malicious, a majority decision won't be possible since replicas will have equal amounts of agreements for each of the values. This will impede system progress.

Another scenario that can undermine the system involves an attacker coordinating nodes to forward incorrect values to applications. As illustrated in Figure 4.9, even though the consensus replicas publish a  $2f + 1$  majority of correct values to the brokers, faults accumulate across each layer transition, ultimately compromising the system's correctness. This issue also underscores how introducing additional communication transitions can negatively impact the system: not only does it increase network complexity, but it also enables further fault accumulation, rendering it impossible to guarantee value correctness. Similar to the previous scenario, the Byzantine model for the number of faulty replicas proves to be insufficient to ensure that the notification value aligns with the original sensor readings.

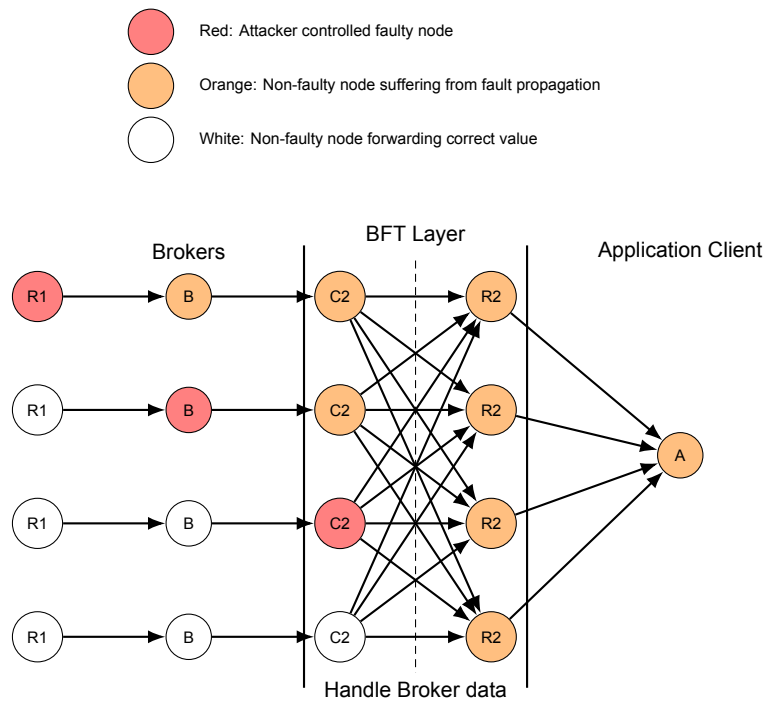


Figure 4.9: Fault propagation scenario 2: Attacker-controlled replica publishes a wrong value in the broker while remaining replicas send the correct value. The same situation happens for both broker and BFT client, meaning wrong values will leak to the next layers. While the consensus majority will be possible, ensuring liveness, the application clients will receive a misleading response, therefore breaking value correctness.

This revision highlights key architectural challenges, particularly those regarding system complexity. By extending replication across all layers, the model overcomes the limitations of broker-only replication. Sensor replication enables the formation of quorums of values, allowing the system to decide based on the majority, while an additional Byzantine consensus layer mitigates inconsistencies in broker state. Anyhow, this architecture still fails against an attacker model where nodes from different layers can collude, even if the Byzantine faulty node threshold is respected. While some solutions, such as [38], reject

the possibility of such a capable attacker, or even couple consensus and broker node faults, as in Trinity [33], we assume such a scenario should be taken into consideration. Not only this, but it is also demonstrated how system resilience and correctness can be threatened by both adding inter-layer message exchanges and/or more communication layers.

### 4.3 Proposed Architecture

The system presented in this work implements a sensor-client-replica-broker network topology that pushes notifications to an app-related BFT client responsible for validating notifications and pushing the result to the application. As such, the latter acts as the interface for applications to interact with the system 4.10.

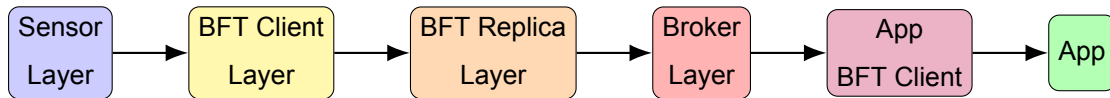


Figure 4.10: Simplified system layout: Sensors send data to BFT clients, which invoke consensus replicas. Replicas publish agreed-upon data to brokers. App clients use BFT clients to read/subscribe, with brokers notifying updates.

It leverages replication to address the sensor fusion problem while also providing Byzantine-level fault tolerance at the broker layer. It draws inspiration from the approach in [39], which separates agreement from execution. As described in Jian Yin et al.'s work, this separation offers advantages over traditional BFT systems by enabling reduced replication at the execution layer. Although our system does not adhere to this approach at the consensus replicas, it emphasizes layer separation to achieve not only a more resilient fault-tolerant model but also to provide greater modularity. Modularity refers to the ability to integrate different implementations of consensus protocols and/or Pub/Sub frameworks. Consequently, while the system was developed using a specific BFT implementation, it may be accommodated to other suitable libraries that provide fault tolerance mechanisms.

Although not explored in this work, additionally, this system could be adapted to CFT models, thereby relaxing replication requirements accordingly.

#### 4.3.1 System Flow

Compared to the naive architecture model, one of the objectives was to understand if the number of replicas could be reduced across sensor and BFT client layers, and based on Marzullo's findings in solving the sensor fusion problem, it shows this is possible with  $2f + 1$  replication [24, 26]. To sustain this, we assume that all values must be bounded within a given predetermined threshold. Readings will only be valid if the given interval is within expectations; otherwise, the reading is not considered. We also extend this level of replication to the BFT client's layer following the work presented by Schneider in [13] to tolerate faulty clients through SMR. Therefore, we reduce the replication level of both these layers while still guaranteeing byzantine tolerance.

Furthermore, the requirement of a second validation layer is dropped by introducing BFT protocol consensus certificates as entity attributes. This cryptographic proof-of-consensus metadata includes a quorum of  $2f + 1$  digital signatures and monotonic counters from participating consensus replicas, enabling downstream fault detection and validation. It enables verifiability successfully dropping the requirement for a second consensus layer, resulting in a simpler network topology - Figure 4.11.

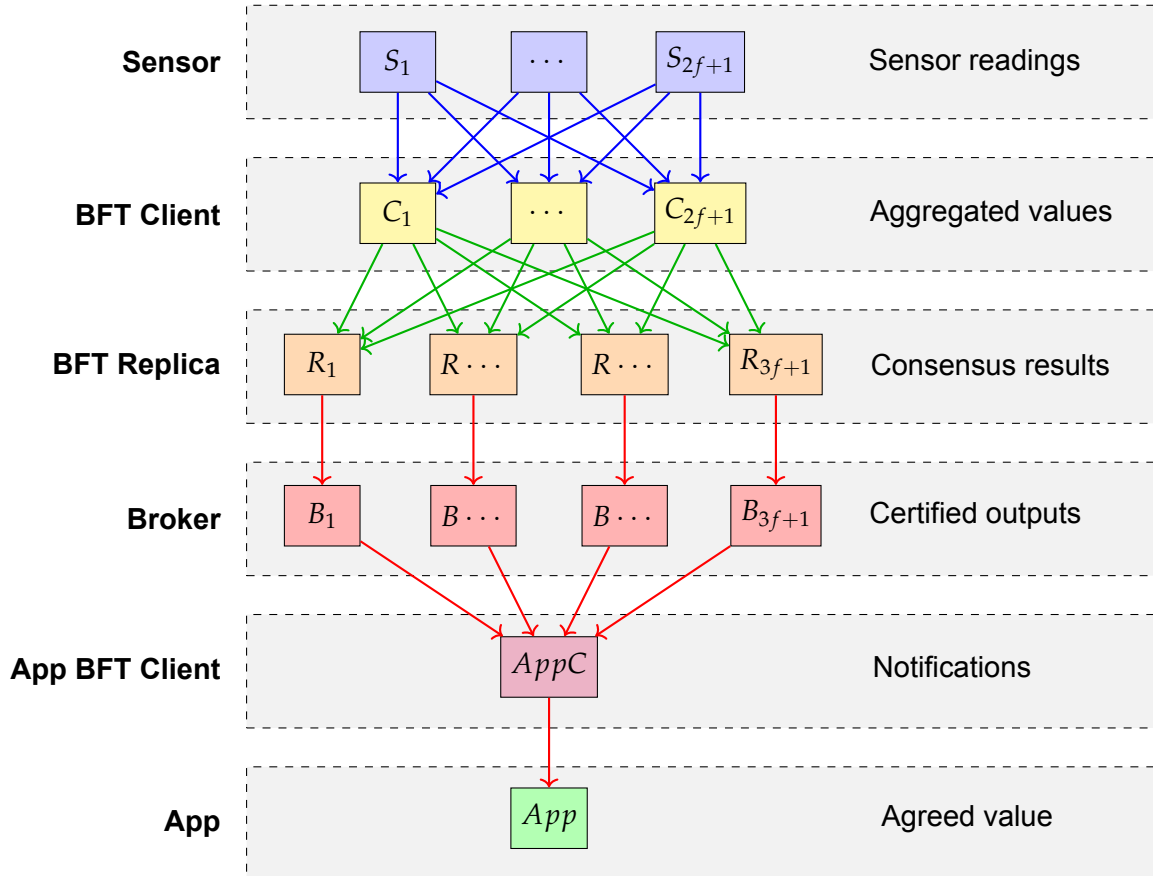


Figure 4.11: BFT IoT Network Architecture Design

Additionally, as shown, the naive approach suffered from fault propagation across layers. The 1-to-1 communication model between layers prohibits recipient non-faulty nodes from detecting value correctness. Therefore, we mask incorrect values by enforcing sensor multicast to clients. While this is not the ideal scenario, provided it increases network bandwidth consumption, we leverage lower levels of replication to balance this out. Furthermore, we introduce the same aggregation process used in **BFT** consensus replicas to allow majority selection, thereby reducing the number of consensus executions and successfully masking faulty values 4.12.

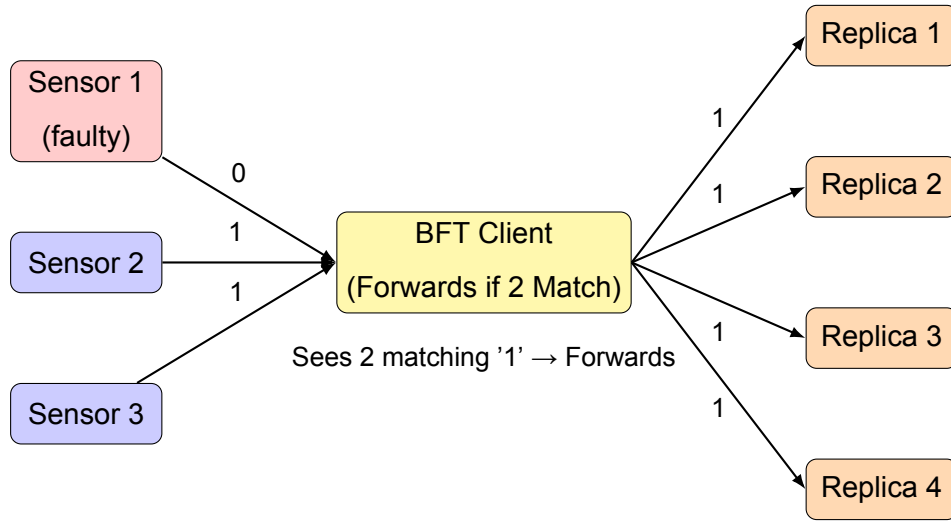


Figure 4.12: Aggregation scenario: For  $f = 1$  faults  $2f + 1$  sensors send each value to each **BFT** client which will aggregate those until a  $f + 1$  threshold is met. If so, clients invoke replica consensus and pass the agreed value. It allows clients to mask erroneous values, impeding fault propagation between sensor-client message exchanges.

For any sensor layer output, each of the  $2f + 1$  **BFT** clients is assumed to receive up to  $2f + 1$  readings. A faulty sensor may deviate by sending inconsistent values to different clients, transmitting incorrect data, or failing to send any data altogether. However, **BFT** clients perform aggregation, which requires a quorum of  $f + 1$  matching readings to initiate consensus. Since at least  $f + 1$  of the  $2f + 1$  sensors are assumed to behave correctly, this mechanism ensures that the majority of valid responses enable the aggregation process to take place, allowing non-faulty clients to converge on the correct value.

These refinements require the extension of **BFT** clients' role within the network: applications must now deploy a **BFT** client to serve as their interface to the system. Apart from previous tasks, each application-related **BFT** client is responsible for aggregating broker notifications until a threshold of  $f + 1$  matching, correctly signed values are received, at

which point the notification is forwarded to the application. The lower threshold requirement is enough to sustain faulty brokers, since the data is verifiable. For any round of execution, if  $f + 1$  values are collected, it guarantees that at least one correct broker has relayed the value proposed by a consensus replica. This enhances system reliability, as the quorum of certificates confirms the value has been agreed upon by a majority of consensus replicas.

#### 4.3.2 Attacker/Fault Model

As shown in previous sections, the new architecture significantly simplifies the network complexity by reducing replication requirements at the sensor-client layers and also completely removing the necessity of a second BFT layer to validate broker values. This is possible by leveraging consensus certificates and by using the client middleware to validate this data. Therefore, we improve the previous fault model by allowing for a more capable attacker.

Nodes can collude in whatever way, provided established Byzantine thresholds are met. We improve upon Trinity's[33] fault model by successfully decoupling inter-layer faults. This means that even if we have faulty components communicating with non-faulty ones, the system continues to make progress by masking malicious values and is able to ensure correctness when values are sent to the client. Figure 4.13 demonstrates the scenario exemplified in 4.8 but with the newer architecture module.

At the broker layer, after consensus is executed, and data contains validation fields, we can assume that even if we have a faulty one, the attacker won't be able to break cryptographic primitives. Therefore, it cannot break quorum signatures and provide a certified wrong value. As such, a non-faulty broker can't receive incorrect values from a malicious replica with valid signatures. We argue that even if broker entity values are misaligned, this will happen at a maximum of  $f + 1$  nodes. Since we have  $3f + 1$  broker replication, it means that for  $f > 1$  the broker layer always has a higher number of correct broker values. However, in the  $f = 1$  scenario, this condition is not met. Anyhow, by assuming brokers cannot break cryptographic primitives, there is no scenario where a wrong value with a forged signature is accepted by a client. We also assume that application-level BFT clients will be aware of the public keys of BFT replicas, allowing signatures to be validated. Figure 4.14 depicts this scenario even in the presence of a multi-layer collusion.

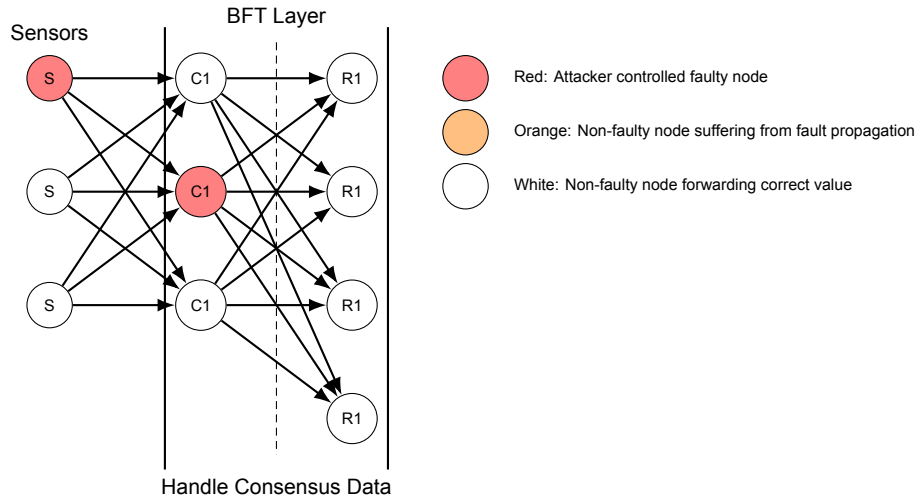


Figure 4.13: Fault propagation mitigation: Attacker-controlled replicas collude to undermine the system's liveness. A malicious sensor purposely sends wrong values to **BFT** clients. Since these requests are aggregated until  $f + 1$  matching values are seen, clients will mask wrong values. Even with a malicious **BFT** client broadcasting the same wrong value to consensus replicas, the process of aggregation is again repeated at the replicas after consensus executions. Since aggregation masks faulty values, the system is able to continue to make progress.

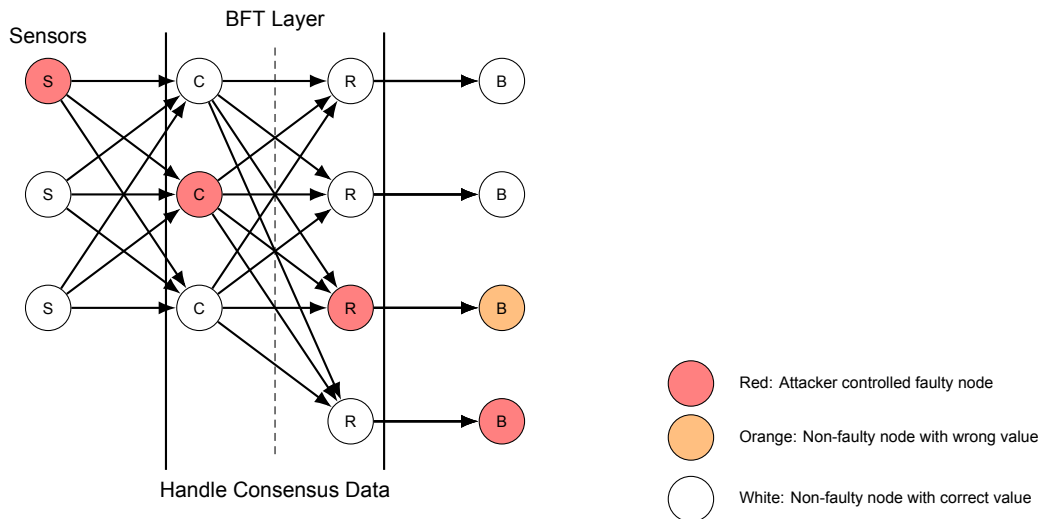


Figure 4.14: Full architecture worst case for  $f = 1$  faulty nodes: Even though fault propagation can happen from replica to broker layers, it is not enough to undermine the system since signatures are not valid. The remaining  $f + 1$  broker values with correct signatures are sufficient to ensure that application clients can accept the response as valid.

Anyhow, the scenario depicted in Figure 4.14 is still exposed to replay attacks. Malicious BFT replicas and Brokers might collude to provide a previously valid value, thereby confusing application BFT clients' aggregation process. Because these only forward messages once they receive  $f + 1$  matching signed values, in a compromised execution, clients would have received two distinct sets of  $f + 1$  matching values that appear valid. While this wouldn't be sufficient to break client execution - the first  $f + 1$  value is taken - it would possibly mean that clients would accept and forward outdated, and therefore incorrect values if received first. To prevent such a scenario, a consensus-related monotonic counter is leveraged to ensure request freshness. This counter is part of the signed messages content and can be validated by clients to ensure freshness. Furthermore, clients maintain a state of last seen highest counter so that it is possible to detect replayed messages and therefore drop them. This effectively prevents attackers from indefinitely replaying outdated messages, enabling clients to detect replay attacks.

#### 4.3.3 Further Assumptions

Apart from previously stated assumptions, this architecture assumes a partial-synchronous network model. It accepts that there is an unknown Global Stabilization Time (GST) where nodes may behave asynchronously. However, after this, any message should be delivered within a given bound, apart from asynchronous notifications. We also assume application BFT clients must know beforehand the public keys of consensus replicas so that consensus certificates can be validated. Additionally, since applications and their BFT clients are expected to run within the same environment, we assume a fault model where if one of these components fails, the whole application won't receive the information. This coupling of faults is considered to be an acceptable assumption, as subscribers are responsible for the software and environment used to query our system. Finally, although attackers may resign from forwarding values to next-layer nodes, they cannot compromise communication channels between honest ones. This means we assume honest-to-honest node communication is delivered within a bound of time, but in the remaining situations, the behavior is arbitrary.

## 4.4 Summary

This chapter begins by examining the limitations of non-replicated and partially replicated Pub/Sub architectures, establishing the motivation for incorporating Byzantine fault-tolerant mechanisms. It then overviews a naive architectural model explaining its components, roles, and highlighting its limitations. This system, which replicates all components in a  $3f + 1$  manner and involves two distinct BFT layers, suffered from significant drawbacks. Its design hindered scalability and proved vulnerable to a more capable attacker model where nodes could collude across layers. Most critically, the 1-to-1 communication between layers led to fault propagation, whereby even if individual layers respected Byzantine thresholds, colluding attackers could prevent majority decisions, jeopardizing the system's Liveness, Safety, and Correctness properties.

To address these limitations, a refined system model is proposed with a simplified sensor-client-replica-broker network topology. This revised architecture reduces sensor and BFT client layer replication to  $2f + 1$ , following Marzullo's work involving the Sensor Fusion problem[24]. Additionally, fault propagation is mitigated by enforcing sensors to multicast to all clients and by incorporating aggregation processes similar to those already used in BFT consensus replicas, masking erroneous values. Another important change is the removal of the second validation layer by leveraging consensus quorum certificates to validate data. This allows application BFT clients to accept values with  $f + 1$  matching valid response certificates and fresh monotonic counter values, preventing replay attacks. The reduction in consensus executions and replication level of sensor and BFT client layers not only reduces the system's network overhead but also possibly lowers hardware requirements. It also supports a more capable attacker model, tolerating cross-layer node collusion as long as Byzantine thresholds are met. This crucially decouples inter-layer faults, dealing with the fault propagation problem of the prior architecture.

## Chapter 5

# Implementation

This chapter outlines details in the practical realization of the proposed **BFT-IoT** system in this dissertation. Building upon the theoretical framework established in Chapter 4, which outlined a more modular architecture and a refined fault model, this section transitions from conceptual design to its tangible implementation. It provides a brief context explaining the adaptations required to adhere to the refined architecture presented in the previous section. Furthermore, it provides an evaluation of system performance and validates the system's fault tolerance capabilities by exploring and demonstrating tested attack scenarios.

### 5.1 Implementation of the Byzantine Middleware Components

This implementation focuses on extending the existing Java client and server frameworks: we use the BFT-SMaRt[16] library to infuse Byzantine fault tolerance into a brokered **Pub/Sub** system. To maintain architectural flexibility, we do not alter BFT-SMaRt itself, but instead, we adapt our middleware layers around it. This design ensures that other fault-tolerant consensus engines or **Pub/Sub** technologies can be adopted in the future with minimal refactoring. Another particular focus of this system is to demonstrate how fault tolerance against Byzantine faults can also be achieved with  $2f + 1$  replication at the sensor and **BFT** clients' layers.

Following FIWARE's frameworks, the given implementation supports the following operations:

**ContextCreate** Creates one or more new context entities in the Context Broker, establishing initial state or introducing new data into the system.

**ContextUpdate** Modifies existing context entities or their attributes—such as appending, updating, or replacing values—by sending an update operation to the Broker.

**ContextQuery** Retrieves the current state of context entities, typically via a synchronous read operation.

**ContextSubscription** Registers interest in specific context changes (e.g., entity updates or creations). The Broker monitors these subscriptions and triggers notifications when conditions are met.

**ContextNotification** An asynchronous message sent by the Context Broker when subscribed context changes occur. This includes both initial notifications (sent on subscription creation) and real-time update notifications.

#### 5.1.1 BFT Broker Proxy

Serving as the server-side Byzantine fault-tolerant replica that receives context operations (create, update, query, subscription) from clients, it processes them through the BFT-SMaRt[16] consensus protocol to achieve total message ordering across consensus replicas and forwards the agreed-upon operations to the underlying FIWARE Context Broker infrastructure. The primary concern with this component was how to adapt it so that consensus information could be shared with other layers, enabling the inclusion of validation mechanisms. Since we assume the correctness of the underlying consensus library, it means that when given consensus-related information, any receiver can confirm that a majority of consensus replicas reached the agreed-upon value. Since we reduce the replication level of sensors and BFT clients, the aggregation threshold drops from  $2f + 1$  matching values to  $f + 1$ . This reduction assumes that out of  $2f + 1$  BFT clients, up to  $f$  may misbehave — e.g., by withholding values or sending inconsistent ones. Requiring only a simple majority of  $f + 1$  matching values still ensures the system can make a correct decision, as there will always be a greater number of matching values:  $f + 1 > f$ .

##### 5.1.1.1 Consensus Quorum Signatures

As mentioned previously, the possibility of validating agreement executions stands as the crucial step to refine the naive architecture and remove the requirement for the second

replication layer. As such, we append consensus metadata into context information to be used later for the validation mechanism.

When **BFT** clients invoke consensus, each non-faulty consensus replica processes it and saves the resulting value until the majority threshold is reached. The last agreement execution that completes this aggregation validation process will trigger the extraction of consensus data using the *MessageContext* class provided by the underlying consensus library. From this, we can extract a set of *ConsensusMessage* objects containing the signed proofs from each participating replica that agreed upon the value, as well as a *MonotonicCounter* value used to ensure request freshness and prevent replay attacks. Since this consensus data is consistent across replicas, it will allow the correct ones to forward consistent context and consensus information to brokers. As such, this quorum of  $2f + 1$  proofs is appended to the context operation data and pushed into each broker accordingly.

With such information, we argue that even if there are malicious colluding attacker nodes across layers, brokers can't send conflicting messages, which are valid. This relies on the fact that the attacker cannot forge proofs, nor can it provide a valid quorum of proofs (from valid replicas) and a valid monotonic counter for a value that non-faulty replicas didn't agree on. Even if brokers try to send conflicting messages, the validation process at the **BFT** clients ensures these are accepted when validating broker notifications.

### 5.1.2 BFT Broker Client

Acting as the client-side interface that triggers the consensus, this client forwards sensor information to be agreed upon. With the contributions in this thesis, its role is further extended to support application client context data management, crucial in removing the necessity of a second consensus layer for notification validation.

#### 5.1.2.1 Aggregation

Following Marzullo's work on the Sensor Fusion problem [24, 26], replication is reduced from  $3f + 1$  to  $2f + 1$  at the sensor and client layers. It was necessary to adapt sensors to broadcast read values so that recipients can mask malicious sensor data. Following the previous architecture, **BFT** clients could receive information from each sensor and forward it to replicas, allowing consensus to run. However, while this would be correct

in terms of Byzantine Fault Tolerance, it would require unnecessarily more consensus rounds to be performed for a single round of sensor readings, increasing overhead and possibly latency. To mitigate this and capitalize on the reduced number of replicas, we introduce the existing **BFT** replica's aggregation process into the **BFT** clients.

Aggregation happens for specific types of messages. It is used in sensor-related operations since these are replicated and provide quorums of messages instead of single requests. As such, for *ContextCreate* or *ContextUpdate* requests, which respectively create and update entity information, aggregation is implemented, thereby reducing the number of consensus invocations while also ensuring fault tolerance. This aggregation is based on evaluating whether values match within a predefined deviation, and if so, they are grouped into a data structure until a threshold of  $f + 1$  valid and matching values is reached. In that case, the **BFT** client will invoke the **BFT** replicas through the *invoke()* method provided by the BFT-SMaRt library.

Additionally, this process is also performed for *ContextNotification* operations. However, such operations are performed by application-related **BFT** clients. Since brokers forward notifications whenever a context update operation is performed, it is mandatory to validate each of them and aggregate matching values. Compared to the aforementioned naive architecture, where applications would have to listen for incoming requests and validate them based on the quorum of received values, we leverage the **BFT** client for such tasks and forward a single message with the aggregated final value. As we assume **BFT** clients and applications run within the same environment(hardware), the fault model decoupling is not taken into consideration here, as both are tied to the same system. The reasoning behind this is that we assume that if an attacker can control the application-related **BFT** client, it means it has access to the underlying machine, meaning it is irrelevant if correct values are forwarded or not. We assume the whole system is compromised on the application side, and so it is not possible to mitigate that scenario through replication of the components outside that system. More concretely, the safety guarantees of the application receiver are a responsibility of that system's maintainer.

#### 5.1.2.2 Quorum Signature Validation

As explained previously, when consensus replicas finish aggregating executions and propagate the information to each context broker, they append consensus information that allows clients to verify the validity and freshness of sent data. As such, the *BFT\_Broker\_Client*

class is extended with a validation method called *verifySigs()*. This includes retrieving the object message, extracting the *validators* field from the payload, and verifying each signature using pre-gathered consensus replicas' public keys. We assume **BFT** clients know each public key from consensus replicas.

If both the monotonic counter and the signatures are valid, the request will be saved into a data structure that aggregates validated messages and forwards them when a threshold of  $f + 1$  matching requests have been collected. If so, it means the client has seen a valid number of matching values with matching signatures validated by a quorum of  $2f + 1$  consensus replicas. The quorum of signatures ensures that at least  $2f + 1 - f = f + 1$  of these signatures come from honest consensus replicas and therefore there is a majority of honest replicas that agreed upon this value.

### 5.1.2.3 Notification Handling

As shown in Figure 4.7, in the previous naive architecture second layer consensus replicas were tasked with notifying Applications that subscribed to a given context entity. This would add more latency to consensus replicas since they were required to listen for incoming notification events, but also would add overhead to the overall system since a single notification event would require  $3f + 1$  brokers to forward the context event to  $3f + 1$  **BFT** clients, which would each trigger a consensus round so that replicas could notify application clients (Figure 5.1).

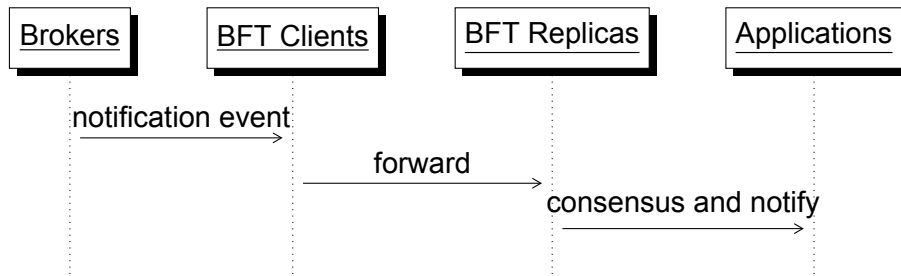


Figure 5.1: Previous Notification Message flow: After an already subscribed context entity is updated, brokers must forward this information to second-layer **BFT** clients. These are responsible for triggering the consensus to validate if the notification majority quorum requirements are met. Each client request will trigger a consensus execution, which will be aggregated until a quorum of  $2f + 1$  matching values is reached. If so, replicas forward the request to the subscribed application(s).

With the new architecture, we extend the role of **BFT** clients to act as the interface for notification processing. A state with the mapping between a given application client and its subscriptions is maintained by the correspondent **BFT** clients, allowing notifications to be forwarded upon broker notification reception.

As explained in the previous subsection, notifications are collected and grouped into batches of matching values until a threshold of  $f + 1$  is reached. This also depends on the validation of the metadata that contains consensus proofs. If successful notifications are forwarded to the application as a single request.

With such changes, even if brokers are Byzantine, we remove the requirement of running another consensus round to validate their notifications (Figure 5.2). This is possible through adding a validation step in the BFT clients, which extracts the signature data from the given notification messages and verifies if the quorum of signatures is valid. Compared to the previous model, we argue that the change could drastically reduce latency, as consensus rounds are considered costly and negatively impact the responsiveness and availability of the whole system.

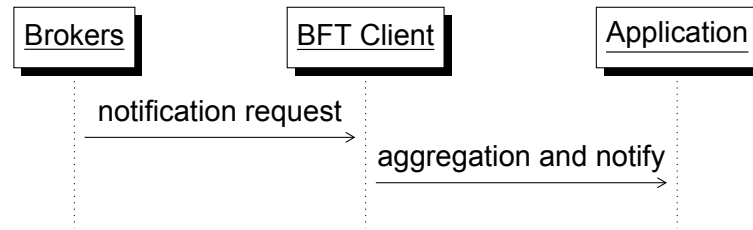


Figure 5.2: New Notification Message flow: After context entity is updated, brokers must forward this information to subscribed application-related BFT clients. These are responsible for validating if the notification majority quorum requirements are met. Each notification takes part in an aggregation process until a quorum of  $f + 1$  matching values is reached with valid signatures and a fresh monotonic counter. If the aggregation process completes successfully, a single notification is forwarded to the application.

## 5.2 Application-level code

The implementation consists of three components: a Node.js sensor workload generator, an application-visualization/subscription server, and a broker HTTP client library. Together, they enable entity registration, updates, queries, and subscriptions for a configurable number of instances, allowing context data provisioning across multiple brokers. The code follows the NGSv2 data model (used in the Orion Context Broker), in which each sensor is represented as an entity with an id, type, and a flexible set of attributes. These attributes are serialized into JSON payloads in accordance with that model.

### 5.2.1 Airquality Sensor Application

At the core is the *AirqualitySensor* class, which encapsulates a sensor's id, type, geolocation, and mutable telemetry attributes (temperature, humidity, pressure, CO2), and

exposes methods to build NGSI v2-compliant JSON payloads for creation and update operations against a broker. A list of coordinates is loaded to instantiate each sensor and maintain a global list to drive concurrent entity creation and periodic updates, effectively simulating a scenario of replicated devices emitting measurements.

#### 5.2.1.1 End-to-End sensor publishing flow

On startup, a JSON file containing latitude and longitude pairs is parsed, and for each entry, an *AirqualitySensor* instance is created with matching randomized readings. It then assembles an entity creation wave by pairing sensors with all broker endpoints and issuing a preliminary *ContextQuery* operation to determine if the entity already exists. For entities not yet present, the client performs a *ContextCreate* with the full entity data. After a short delay to allow registrations to settle, the system continuously generates new sensory data and applies it to all sensors, emitting updates to every broker on a fixed cadence (two seconds by default). Each produces a set of asynchronous *ContextUpdate* operations carrying the new sensor reading values.

#### 5.2.2 Application

The application is a Node.js service that handles *ContextSubscription* operations and exposes endpoints for receiving asynchronous notifications. It also runs a live dashboard to visualize incoming sensory data notifications. On startup, it launches an *Express* server to listen for notification events and dashboard requests, and it issues a *ContextSubscription* request to register itself as the notification target. To manage bursts of high-rate notifications, the service uses a queue: the *airqualityHandlerQueued* method enqueues each incoming event and triggers a queue handler, which checks if processing is already underway. If not, it processes the next event in order, validates and saves it in the application state, and then broadcasts the update to all connected clients via server-sent events.

### 5.3 Summary

This chapter details the practical realization of the proposed system, building directly on the theoretical framework and refined fault model presented in Chapter 4. The system extends existing Java client and server frameworks, leveraging the BFT-SMaRt library to embed BFT consensus capabilities into an IoT Pub/Sub system. Crucially, the implementation ensures architectural flexibility by adapting middleware layers around BFT-SMaRt,

rather than modifying the library itself, which allows for future integration of other consensus engines or **Pub/Sub** technologies. This implementation effectively mitigates fault propagation, even against colluding attackers across layers, by leveraging consensus certificates and client-side validation, provided Byzantine thresholds are met. The introduction of monotonic counters further enhances security by preventing replay attacks. This refined architecture ultimately leads to a more modular design, potentially reduced network complexity, improved system performance, and availability.

The implementation also includes a mock sensor-request generator and a subscriber application to simulate real-world operations: the generator allows publishing sensor data assuming a replicated environment, while the subscriber application consumes entity updates via subscriptions. Together, they enable a full end-to-end exploration of the **Pub/Sub** flow and form the basis for both the performance testing and system-correctness validation work described in the upcoming sections.

## Chapter 6

# Results

To validate the correctness of our system and demonstrate the cost of implementing this level of fault tolerance, different attack scenarios are simulated to evaluate performance degradation/impact and understand how the system behaves under attacks compared to non-replicated architectures or non-faulty replicated ones. This section describes multiple tested attack scenarios and provides data to justify the described improvements for the refined architecture.

### 6.1 Experimental Setup

#### 6.1.1 System Environment

The experimental environment consists of applications deployed in Docker[40] containers that share the same virtual network, orchestrated through `docker-compose` definitions. All experiments were executed on a macOS system equipped with a ten-core Apple M4 processor and 16 GB of RAM. This setup ensures a controlled and reproducible environment for evaluating the system's performance.

#### 6.1.2 Data Collection and Metrics Extraction

Network traffic data was captured during each experimental run using Wireshark[41], producing packet trace files in the `pcapng` format. These traces were subsequently processed to extract relevant performance metrics, such as throughput and latency. The extraction and aggregation of metrics were automated through a Python script, which averages results across independent repetitions for each experiment. The aggregated data was

stored in CSV files, serving as the basis for the graphical plots presented in the upcoming sections(Figure 6.1).

### 6.1.3 Workload and Scenarios

The initial evaluation covers two representative scenarios: (i) a sensor publish scenario, in which sensory data is disseminated through the system, including operations of context creation, context query, and context update; and (ii) an end-to-end scenario, where sensors publish context data and brokers disseminate updates to subscribers. This scenario extends the previous setup by adding four rounds of context updates, performed at two-second intervals, along with context notifications and the execution of a context subscription call. Each testing scenario was executed in batches to ensure statistical robustness. To perform the attack scenarios, alternative classes were implemented, thereby enabling the assessment of system behavior under adverse conditions. The attack scenarios are evaluated only for scenario *ii* as it provides a clearer and more relevant depiction of the overall system performance.

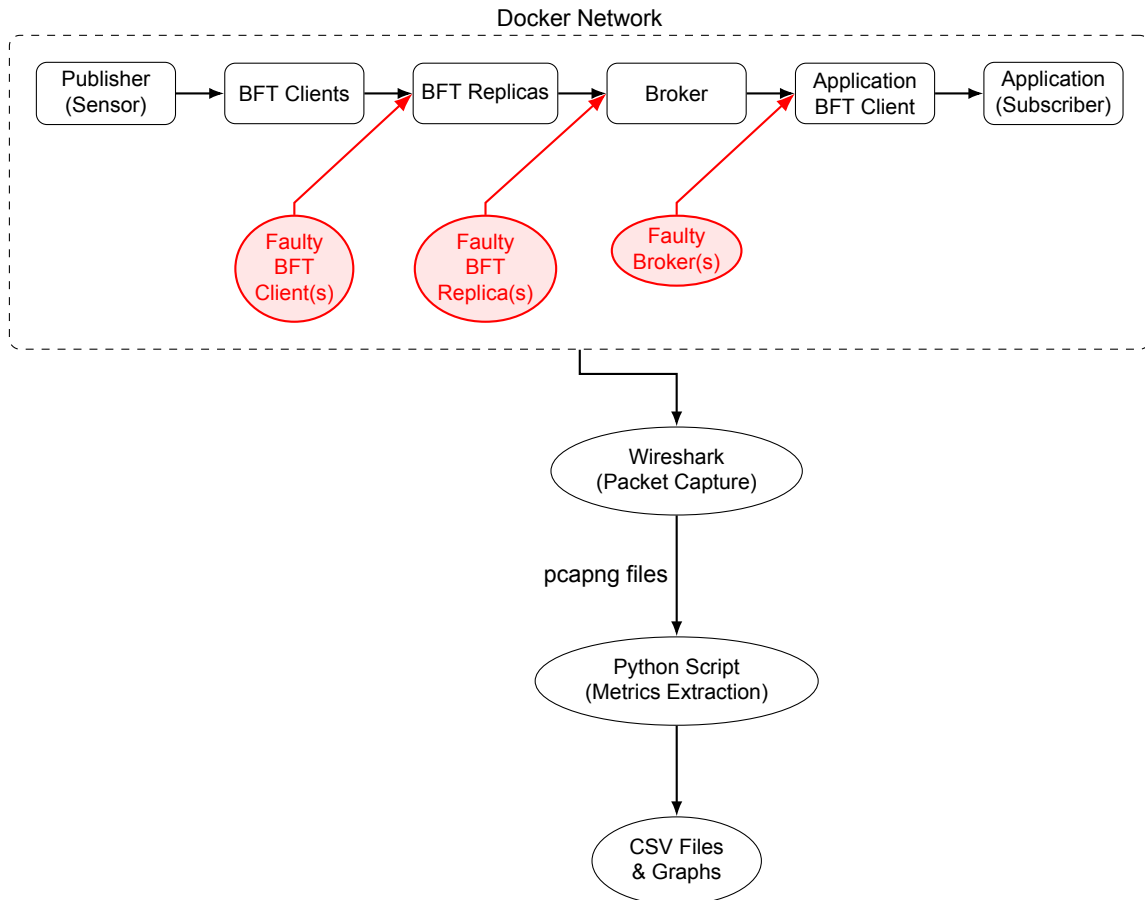


Figure 6.1: Experimental setup showing data flow and internal faulty components used to mimic malicious behavior in replicated architectures.

## 6.2 Non-replicated system

To establish a baseline for the performance evaluation, we first consider a traditional non-replicated system architecture, in which no Byzantine fault tolerance mechanisms are implemented. In this setup, all components operate as single instances, and no attempts were made to simulate attacks or fault scenarios as these architectures are not resilient against them. The purpose of these tests is not to assess resilience, but rather to measure the performance of the system in a non-byzantine architecture scenario.

By establishing this baseline, we can later understand the impact of using replication and fault-tolerance mechanisms. The comparison to the non-replicated allows for a transparent display of the trade-offs introduced by replication and consensus mechanisms.

### 6.2.1 Sensor(Publisher) flow

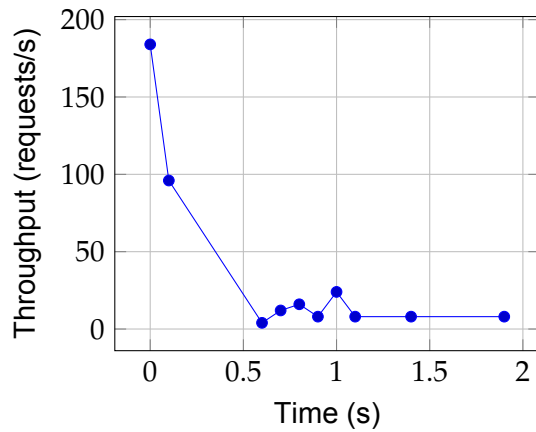


Figure 6.2: Average Throughput over time

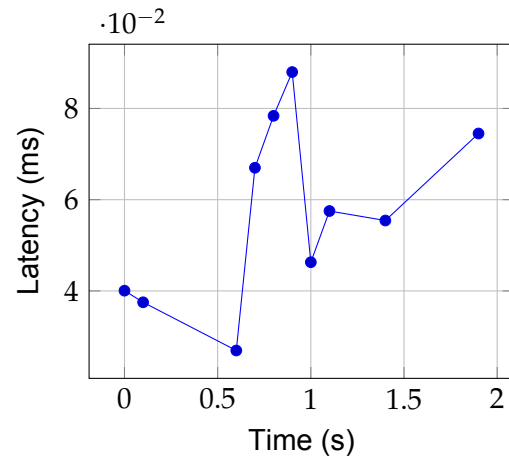


Figure 6.3: Average Latency over time

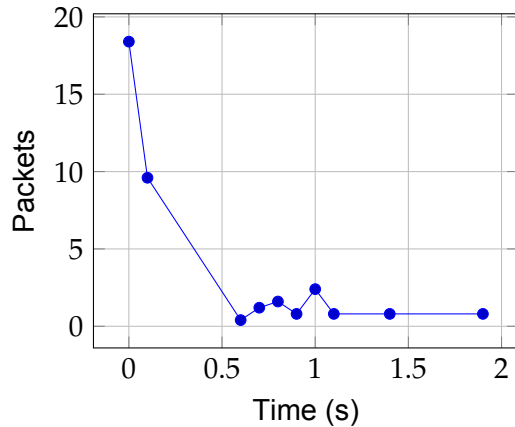


Figure 6.4: Average Packet Count over time

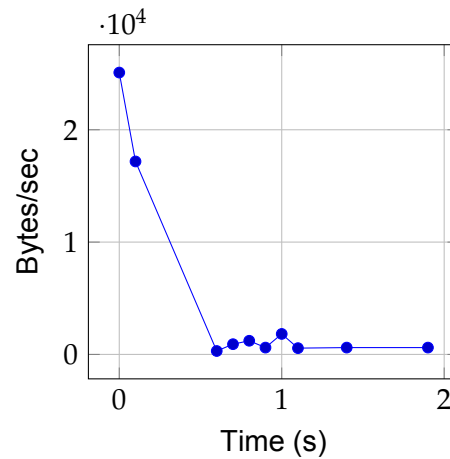


Figure 6.5: Average Bytes over time

## 6.2.2 End-to-End flow

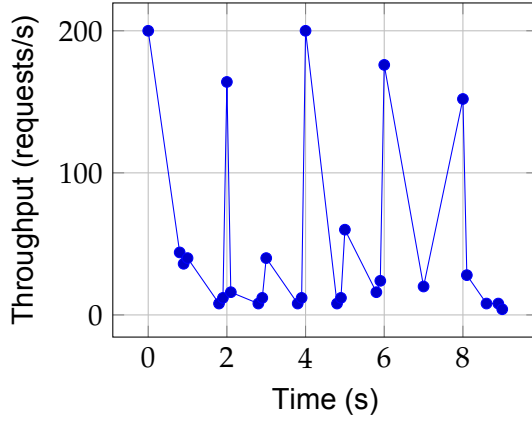


Figure 6.6: Average Throughput over time

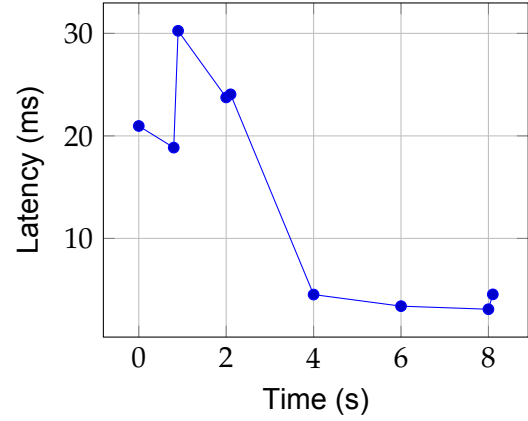


Figure 6.7: Average Latency over time

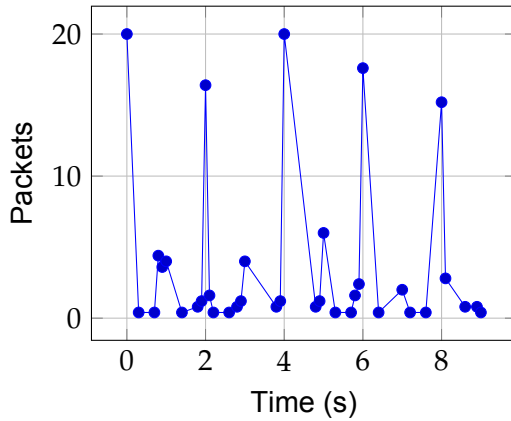


Figure 6.8: Packet Count over time

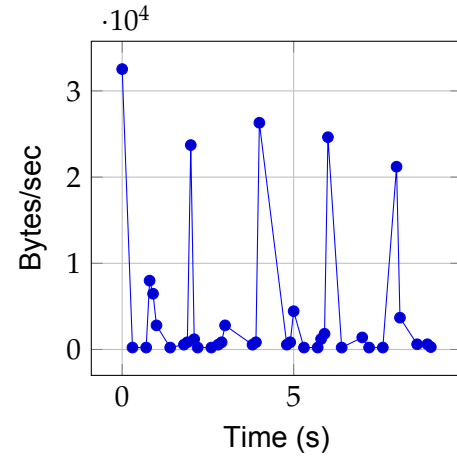


Figure 6.9: Amount of Bytes over time

The non-replicated configuration establishes the baseline performance of the IoT architecture under crash-free operating conditions. Comparing the data in Figures 6.2 to 6.5 with Figures 6.6 to 6.9 reveals consistent performance patterns, as the spikes in throughput and latency are comparable between the publisher and the end-to-end flows. As expected, the results indicate that context update operations account for the highest resource consumption. This is expected since it triggers evaluation of subscriptions, notification generation and delivery mechanisms, and database write operations. Nevertheless, the system maintains responsiveness, exhibiting consistently low latency values.

These quantified performance levels are essential, as they will allow the subsequent sections to accurately measure the impact of BFT mechanisms, thereby clearly displaying the trade-offs introduced by replication and consensus on these key performance metrics.

## 6.3 Replicated system: No Faults

Building upon the baseline established with the non-replicated configuration, we next evaluate the replicated system architecture that incorporates Byzantine fault-tolerant mechanisms. In this setup, replicated components are deployed to ensure system correctness even in the presence of malicious or faulty nodes.

By comparing the performance metrics of the replicated system against the non-replicated baseline, we can demonstrate the overhead introduced by replication and assess how fault-tolerance mechanisms affect throughput, latency, and resource consumption. This comparison provides valuable insights into the trade-offs between system reliability and performance, and highlights the practical implications of deploying Byzantine fault-tolerant architectures in real-world applications.

### 6.3.1 Sensor(Publisher) flow

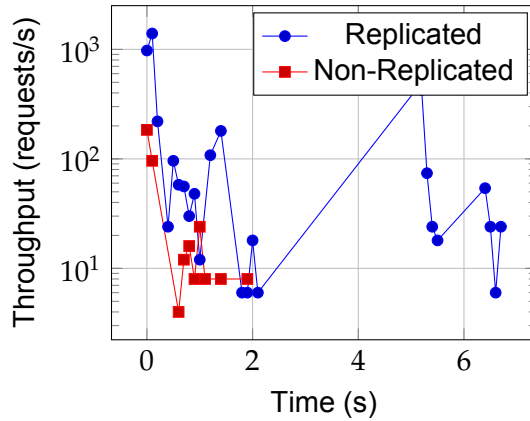


Figure 6.10: Average Throughput over Time

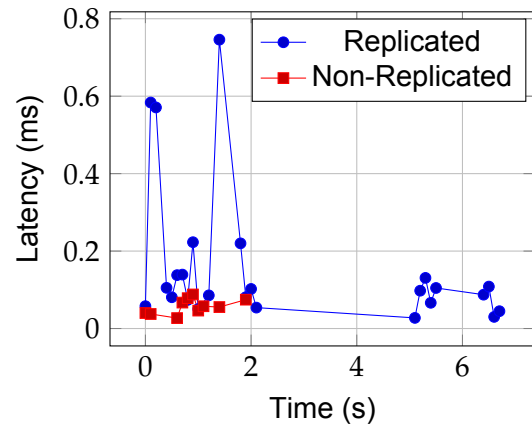


Figure 6.11: Latency over Time

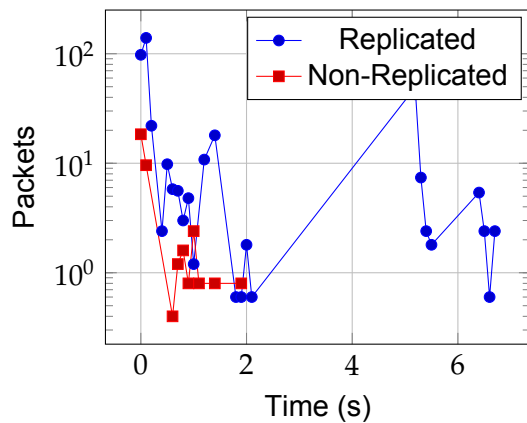


Figure 6.12: Average Packet Count

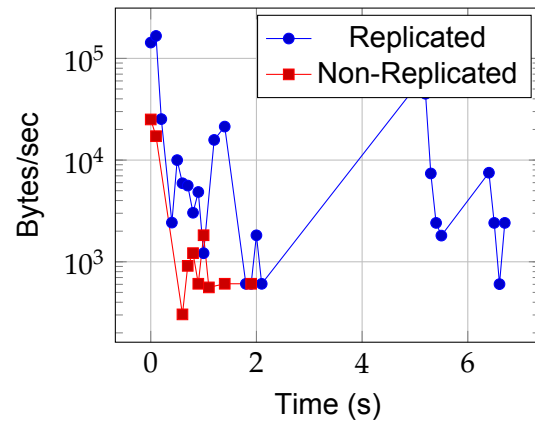


Figure 6.13: Average Bytes per Second

Comparing the non-replicated and replicated systems reveals not only higher bandwidth consumption - indicated by increased packet counts and bytes sent - but also elevated latency. Figures 6.10 and 6.11 illustrate these trends and show that the publisher flow takes longer to complete in the replicated scenario. The introduction of consensus mechanisms adds delay as values must be agreed upon before proceeding. As such, it shows a direct trade-off between fault tolerance and performance. Moreover, the addition of replicated nodes produces more network overhead: as Figure 6.12 illustrates, in the replicated system, the number of packets increased by an order of magnitude compared to the non-replicated case.

### 6.3.2 End-to-End flow

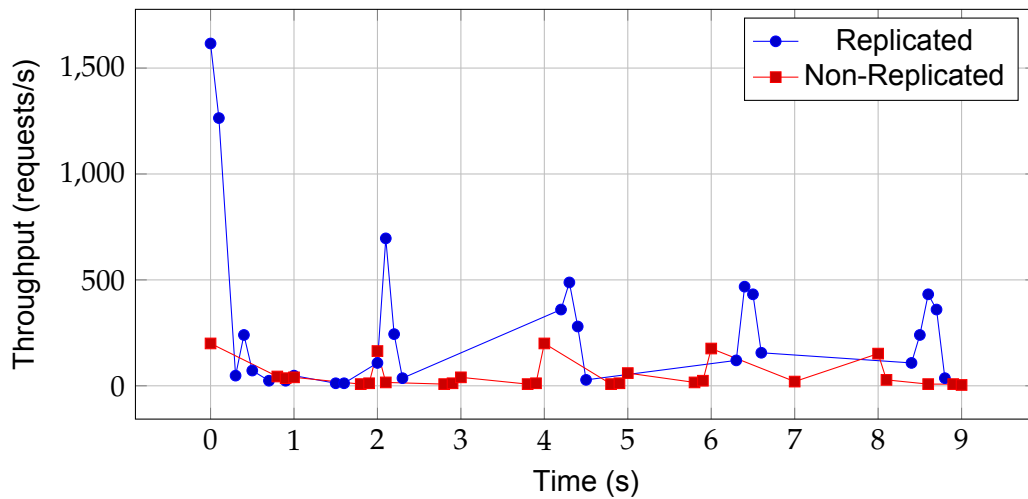


Figure 6.14: Average Throughput over Time

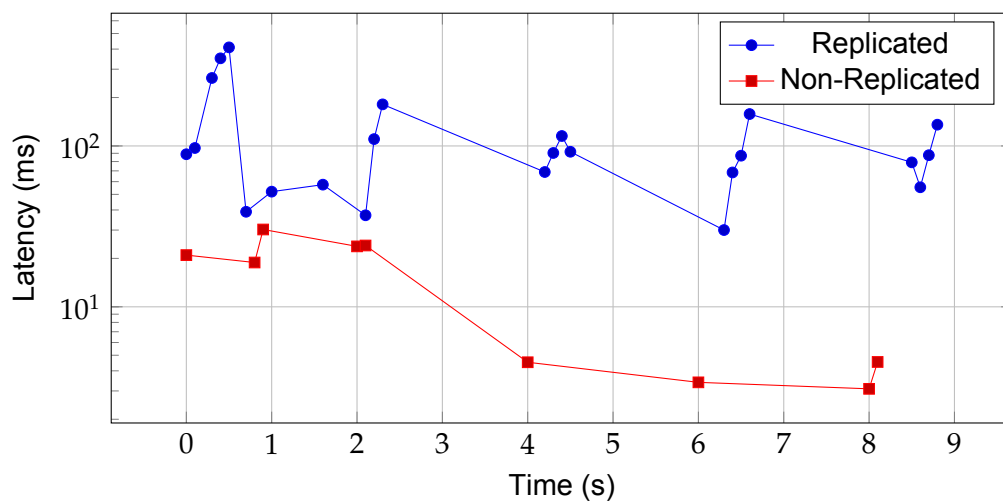


Figure 6.15: Average Latency over Time

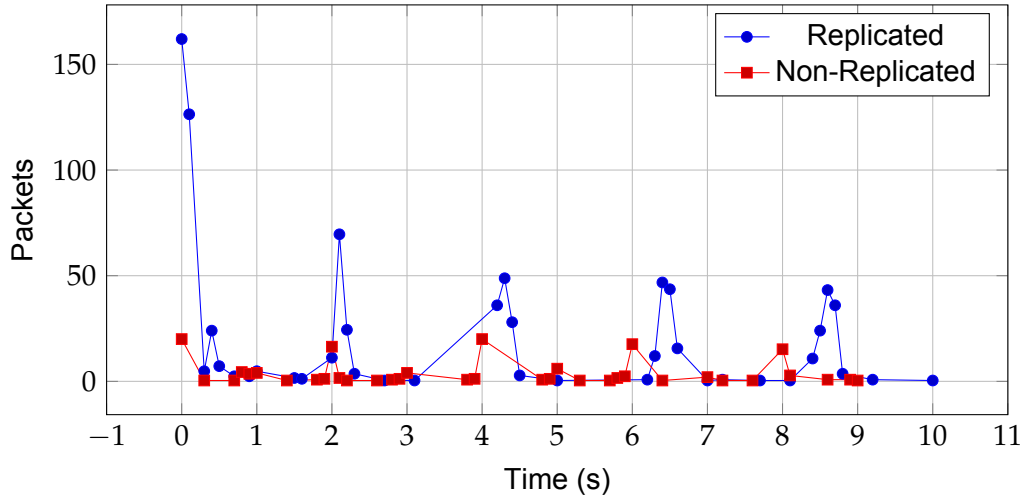


Figure 6.16: Average Packet Count

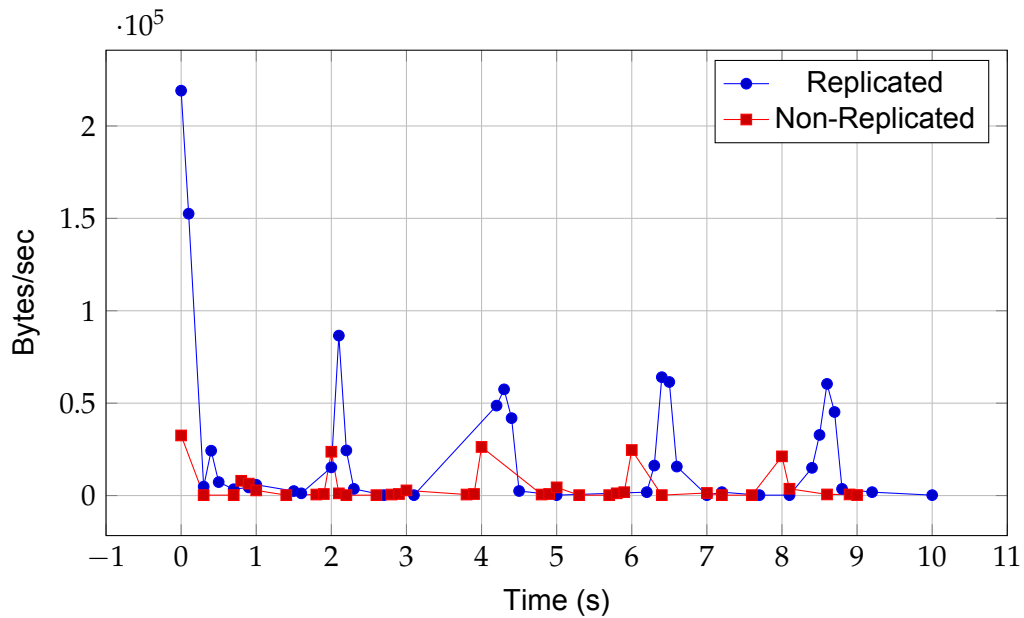


Figure 6.17: Average Bytes per Second

In the end-to-end flow, the same trend is apparent: Figure 6.14 highlights increased network overhead via pronounced peaks. As more nodes broadcast requests, throughput also surges sharply. During the initial phase, variance is highest: nodes issue context read requests to verify if the entity exists and then proceed with creation if it doesn't, followed by update operations. These operations are issued every two seconds, resulting in more consistent throughput peaks. Nevertheless, overall performance degrades due to rising bandwidth demands and network overhead, resulting in a significant latency increase. Figure 6.15 depicts this increase of roughly one order of magnitude compared to the non-replicated system. Furthermore, Figure 6.17 displays a marked increase in bytes

transmitted. This rise stems not only from a larger packet count but also from the enriched payloads: validation metadata is appended both to replica-broker and to notification requests, increasing message size.

These results reveal that replication and consensus mechanisms incur a tangible performance cost. Message payload size increases because validation metadata is appended to replica and notification messages, further amplifying bandwidth and processing load. These results illustrate the inherent trade-off: improved fault tolerance and consistency come at the expense of performance and efficiency.

## 6.4 Attacker Simulations

This section presents the results from the validation phase of our proposed **BFT** architecture, conducted under attacker-simulation experiments. This testing phase focused on validating the correctness of the whole system when mitigating the fault scenarios described in section 4.2.3. Concretely, it outlines the performance comparison between the non-faulty scenario and under arbitrary faults to demonstrate the impact on performance obtained by our practical implementation.

As such, this section focuses on establishing a comparison to the baseline scenario where no faults are present in the system and all tests were performed assuming the end-to-end flow, as it provides a clearer understanding both in terms of performance and system correctness.

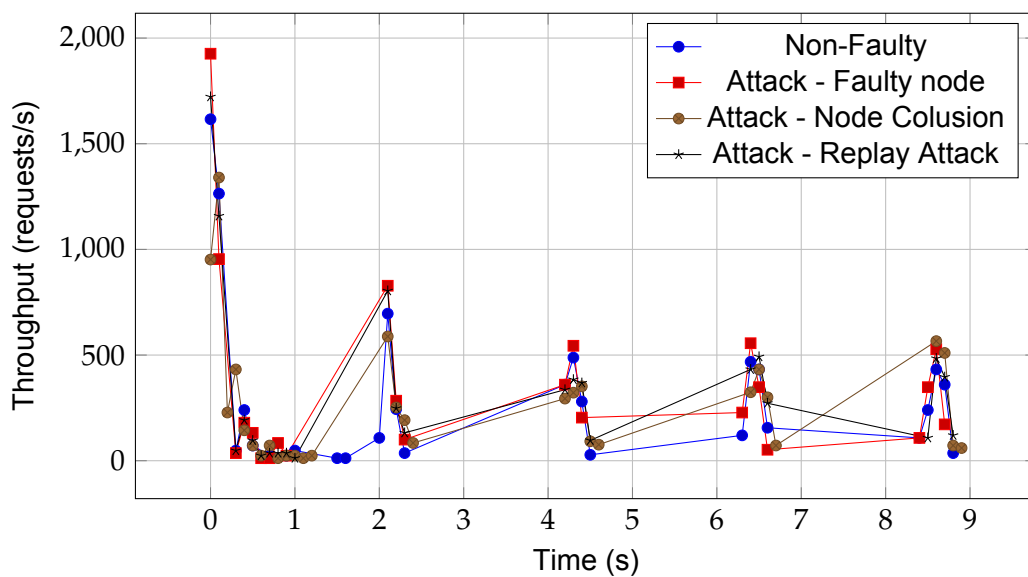


Figure 6.18: Average Throughput over Time

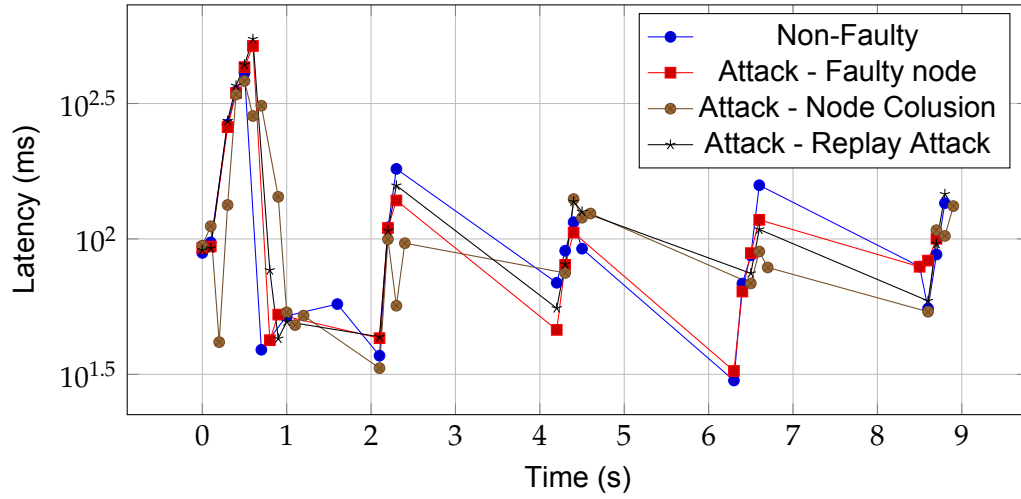


Figure 6.19: Latency over Time

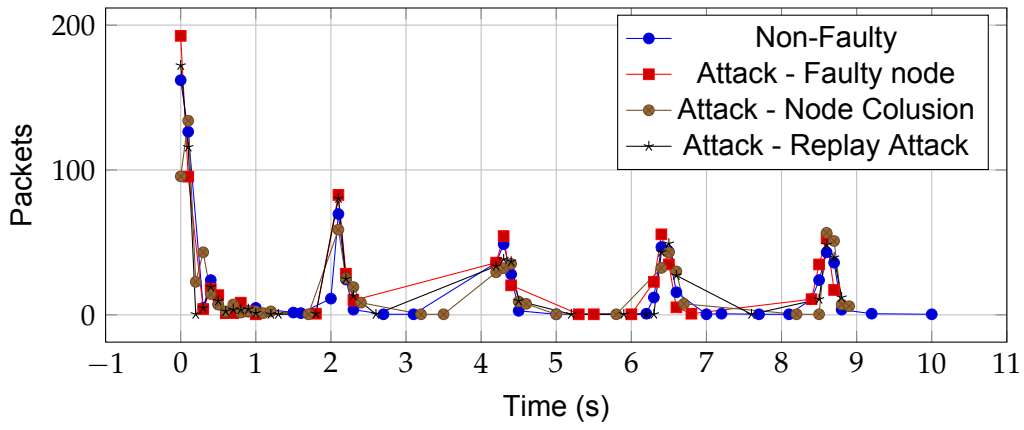


Figure 6.20: Average Packet Count

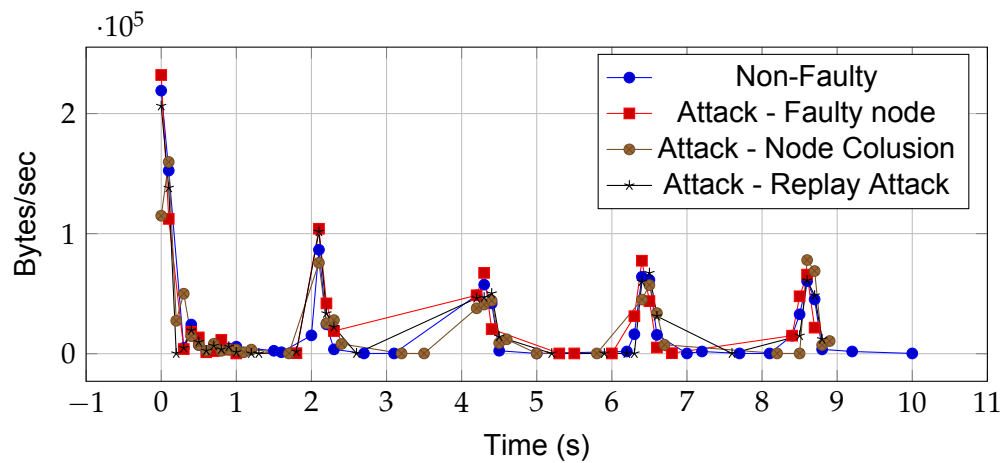


Figure 6.21: Average Bytes per Second

From the execution of our tests, it was possible to confirm that the strategies applied to mitigate the fault scenarios ensure the correctness of the system while upholding the

availability and consistency requirements of **BFT** systems. Furthermore, as shown in Figures 6.18 to 6.21, the performance degradation was shown to be negligible for all test scenarios. This was expected since the amount of data exchanged within the system doesn't vary significantly. Considering a scenario where all nodes keep forwarding data, even if malicious/faulty, the consensus and validation processes still execute successfully, providing a comparable level of overhead.

Although subject to Distributed Denial of Service (**DDoS**) attacks, this was not considered in our test cases since it could be mitigated with mechanisms such as rate-limiting and/or throttling. Anyhow, this can significantly impact overall system performance for a given window of attack. However, even if experiencing a temporary degradation in performance, the correctness properties of the system still hold, as consensus execution and notification validation processes would eventually resume.

## 6.5 Summary

The evaluation of the system's performance began by establishing a baseline by using a traditional non-replicated **IoT** architecture, where no **BFT** mechanisms were implemented. As expected, this setup exhibited the highest performance, characterized by the lowest latency, since it operated without the overhead associated with redundancy or consensus protocols.

The introduction of the proposed **BFT** architecture, even under non-faulty conditions, revealed a substantial performance cost compared to the baseline system. The results consistently showed that **BFT** mechanisms introduce substantial latency. For the end-to-end flow, latency increased by approximately one order of magnitude compared to the non-replicated system. This degradation occurs because the consensus mechanisms necessary for fault tolerance perform multi-phase broadcast message exchange processes where values must be agreed upon before the execution can proceed, inherently adding delay. This comparison clearly displayed the trade-off inherent in deploying fault-tolerant architectures, where improved reliability comes at the expense of performance efficiency. This resource consumption is amplified by the required validation mechanisms; message payload size increases significantly, related to the injection of quorum cryptographic validation metadata, which must be appended to replica and notification requests, further increasing bandwidth and processing load.

Finally, performance assessments conducted under various simulated attack scenarios demonstrated the robustness of the architecture. When comparing the system's performance under malicious conditions against the performance of the non-faulty replicated system, the degradation in performance was found to be negligible across all attack scenarios. This scenario was expected because the underlying overhead is primarily caused by the required **BFT** consensus and validation processes, which successfully execute even when nodes are malicious to maintain consistency and availability. The results ultimately confirmed that the strategies applied successfully mitigated malicious behavior, ensuring the correctness of the system while maintaining the **BFT** requirements for availability and consistency, thereby potentially justifying the performance cost incurred by the replication mechanisms.

## Chapter 7

# Conclusion

This dissertation successfully addressed the critical challenge of ensuring liveness and correctness in IoT systems, by proposing and validating a multi-layered architecture that integrates BFT into the Pub/Sub model. Traditional IoT architectures, often reliant on non-replicated components, proved vulnerable to non-benign failures, including malicious or arbitrary component behavior and the presence of SPoF. The central objective was to ensure data consistency (Safety) and availability (Liveness), even when facing these Byzantine threats.

### 7.1 Implemented system

The implemented system validates the theoretical premise that resilience can be significantly enhanced while mitigating the substantial overhead typically associated with full replication. A key architectural improvement was the reduction of replication to  $2f + 1$  nodes at the sensor and BFT client layers, inspired by solutions to the Sensor Fusion problem, thereby preserving Byzantine properties while decreasing network overhead and costs compared to the standard  $3f + 1$  requirement.

Additionally, the refined architecture successfully mitigated the risk of fault propagation across layers - a vulnerability identified in naive multi-layer BFT systems. This, however, was achieved by mandating that sensors broadcast their readings directly to BFT clients, allowing these clients to perform an initial aggregation step that effectively masks erroneous values before invoking consensus. This strategy ensured that the system could tolerate cross-layer node collusion, upholding Liveness and Safety, but with the inherent network overhead cost of broadcasting values.

Furthermore, the system was significantly optimized by eliminating the need for a second **BFT** consensus layer for notification validation. Instead, consensus validity was enforced by embedding cryptographic proof-of-consensus metadata —specifically, a quorum of  $2f + 1$  signatures and a monotonic counter — directly into the data disseminated by the brokers. Application-level **BFT** clients validate this metadata to verify the notification’s authenticity and freshness, a mechanism confirmed to prevent replay attacks. Experimental results confirmed the system’s robustness under various attack scenarios (Faulty Node, Node Collusion, and Replay Attack), demonstrating that performance degradation was negligible compared to the non-faulty replicated system, thus confirming the successful balance between resilience and operational performance.

## 7.2 Limitations

While effective in achieving **BFT** properties, the system evaluation highlighted inherent limitations primarily related to performance costs. The results conclusively demonstrated that the implementation of **BFT** mechanisms introduces significant latency when compared to the non-replicated baseline architecture. As demonstrated in the end-to-end flow tests, latency increased by approximately an order of magnitude across all scenarios. The required replication and the injection of validation metadata (signatures and monotonic counters) significantly increase message payload sizes, amplifying network bandwidth demands and processing load compared to the non-replicated system. Also, the packet count increased by an order of magnitude even in non-faulty replicated scenarios.

Furthermore, our network model assumes that sensors publish data roughly synchronously. While this may match certain environments or sensor capabilities, it introduces a dependency: each consensus execution must complete before the next publish event. If it does not, the system stalls, causing increased latency. In scenarios with higher replication, this effect can become more pronounced as consensus delay may increase.

## 7.3 Future Work and Improvements

Based on the limitations identified and the ongoing improvement of **BFT** technologies, several directions for future research can be pursued to further improve the system’s efficiency and capabilities. One potential approach would be to design a mechanism where only a single broker sends the full notification response, while the receiver acknowledges this message to all other brokers. If an acknowledgment fails, the remaining brokers could

retransmit the message. This strategy could significantly decrease the number of redundant messages while still maintaining system resilience and reliability. Additionally, the introduction of proactive recovery mechanisms could be explored to address cases where the number of faulty nodes exceeds the tolerable limit. Investigating strategies that allow the system to gracefully degrade or recover quickly when such conditions occur would improve fault resilience. This might include techniques for early detection of node corruption, dynamic reconfiguration, or automated node replacement. Finally, future research could also explore the adoption of more advanced **BFT** protocols to enhance scalability and performance. In particular, incorporating trusted hardware was shown to reduce the required replica count from  $3f + 1$  to  $2f + 1$ , thereby lowering latency and hardware overhead while preserving safety and liveness[42, 43]. Investigating practical deployment of such architectures in **IoT** environments — with constrained resources and high update rates — could contribute significantly to improving system efficiency and reducing resource footprint.

# References

- [1] A. Rayes and S. Salam, *Internet of Things (IoT) Overview*. Springer, 2017. [Cited on page 1.]
- [2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015. [Cited on page 1.]
- [3] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in internet of things: The road ahead," *Computer Networks*, vol. 76, pp. 146–164, 2015. [Cited on page 1.]
- [4] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, p. 7, 2009. [Cited on page 1.]
- [5] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982. [Cited on pages ix, 1, 2, 6, 7, and 8.]
- [6] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI'99: Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 173–186. [Cited on pages 1, 8, 9, 12, 13, and 19.]
- [7] Y. Yu, Y. Zhang, and X. Wang, "A holistic analysis of internet of things (iot) security: Principles, practices, and new perspectives," *Future Internet*, vol. 16, no. 2, p. 40, 2022. [Online]. Available: <https://www.mdpi.com/1999-5903/16/2/40> [Cited on page 4.]
- [8] OASIS. (2025) Mqtt: The standard for iot messaging. Accessed: 2025-01-15. [Online]. Available: <https://mqtt.org/> [Cited on page 5.]

- [9] I. Kamil and M. Abdulsalam, "Principles of fault tolerance in it systems: A review," *International Journal of Computer Sciences and Engineering*, vol. 11, no. 4, pp. 1–9, 2024. [Online]. Available: <https://doi.org/10.14445/23488387/IJCSE-V11I4P101> [Cited on page 5.]
- [10] M. Marcozzi, O. Gemikonakli, E. Gemikonakli, E. Ever, and L. Mostarda, "Availability evaluation of iot systems with byzantine fault-tolerance for mission-critical applications," *Internet of Things*, vol. 23, p. 100889, 2023. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2542660523002123> [Cited on page 5.]
- [11] V. Hadzilacos, "Connectivity requirements for byzantine agreement under restricted types of failures," *Distributed Computing*, vol. 2, no. 2, pp. 95–103, 1987. [Online]. Available: <https://doi.org/10.1007/BF01667081> [Cited on page 6.]
- [12] R. A. Bazzi and G. Neiger, "Simplifying fault-tolerance: providing the abstraction of crash failures," *J. ACM*, vol. 48, no. 3, p. 499–554, May 2001. [Online]. Available: <https://doi.org/10.1145/382780.382784> [Cited on page 6.]
- [13] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, Dec. 1990. [Online]. Available: <https://doi.org/10.1145/98163.98167> [Cited on pages 8 and 39.]
- [14] F. Pelekoudas-Oikonomou, G. Zachos, G. Mantas, J. Ribeiro, J. M. C. Bastos, and J. Rodriguez, "A scalable approach of practical byzantine fault tolerance algorithms for iomt blockchains," in *2022 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, 2022, pp. 124–129. [Cited on page 9.]
- [15] GeeksforGeeks. (2021) Practical byzantine fault tolerance (pbft). Accessed: 2025-02-13. [Online]. Available: <https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/> [Cited on pages ix and 9.]
- [16] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362. [Cited on pages ix, 9, 11, 12, 13, 19, 33, 34, 45, and 46.]
- [17] D. Mazières. (2015) Safety vs liveness. Accessed: 2025-02-17. [Online]. Available: <https://www.scs.stanford.edu/~dm/blog/safety-vs-liveness.html> [Cited on page 9.]

- [18] I. Yanakiev. (2018) Federated byzantine agreement. Accessed: 2025-02-13. [Online]. Available: <https://medium.com/data-science/federated-byzantine-agreement-24ec57bf36e0> [Cited on page 9.]
- [19] S. D. Foundation. (2015) On worldwide consensus. Accessed: 2025-02-13. [Online]. Available: <https://stellar.org/blog/foundation-news/on-worldwide-consensus> [Cited on pages ix and 9.]
- [20] FIWARE Foundation. (2011) What is fiware? Accessed: 2025-03-22. [Online]. Available: <https://www.fiware.org/2011/05/17/what-is-fiware/> [Cited on page 10.]
- [21] ——. (2025) Fiware – open source platform for smart solutions. Accessed: 2025-03-22. [Online]. Available: <https://www.fiware.org/> [Cited on page 10.]
- [22] D. M. Berbes Villalón, L. Sánchez Jiménez, M. de la Iglesia Campos, M. E. Díaz Aguirre, and T. Delgado Fernández, “Una arquitectura iot para ciudades inteligentes basada en la plataforma fiware,” *Revista De Ciencia Y Tecnología*, vol. 38, no. 1, pp. 20–27, 2022. [Online]. Available: <https://doi.org/10.36995/j.recyt.2022.38.003> [Cited on page 10.]
- [23] S. S. Iyengar and R. R. Brooks, *Fundamentals of Brooks–Iyengar Distributed Sensing Algorithm*. Cham: Springer, 2019. [Online]. Available: <https://link.springer.com/book/10.1007/978-3-030-33132-0> [Cited on pages 13, 14, 17, and 18.]
- [24] K. Marzullo, “Implementing fault-tolerant sensors,” NASA, Tech. Rep. NASA-CR-186251, May 1989, nASA Contractor Report. [Online]. Available: <https://ntrs.nasa.gov/citations/19900006084> [Cited on pages 14, 17, 34, 39, 44, and 47.]
- [25] R. Ivanov, M. Pajic, and I. Lee, “Attack-resilient sensor fusion,” in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014, pp. 1–6. [Cited on pages ix, 14, and 15.]
- [26] K. Marzullo, “Tolerating failures of continuous-valued sensors,” *ACM Trans. Comput. Syst.*, vol. 8, no. 4, p. 284–304, Nov. 1990. [Online]. Available: <https://doi.org/10.1145/128733.128735> [Cited on pages ix, 15, 17, 18, 39, and 47.]
- [27] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, “Reaching approximate agreement in the presence of faults,” *J. ACM*, vol. 33, no. 3, p. 499–516, May 1986. [Online]. Available: <https://doi.org/10.1145/5925.5931> [Cited on pages ix, 15, and 16.]

- [28] S. R. Mahaney and F. B. Schneider, “Inexact agreement: accuracy, precision, and graceful degradation,” in *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 237–249. [Online]. Available: <https://doi.org/10.1145/323596.323618> [Cited on pages ix, 16, 17, and 18.]
- [29] R. Brooks and S. Iyengar, “Robust distributed computing and sensing algorithm,” *Computer*, vol. 29, no. 6, pp. 53–60, 1996. [Cited on pages ix, 17, and 18.]
- [30] T. Chang and H. Meling, “Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure (position paper),” 10 2012. [Cited on pages 18 and 20.]
- [31] R. Garcia, R. Rodrigues, and N. Preguiça, “Efficient middleware for byzantine fault tolerant database replication,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 107–122. [Online]. Available: <https://doi.org/10.1145/1966445.1966456> [Cited on pages v, ix, 19, 20, and 29.]
- [32] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” *SIGMOD Rec.*, vol. 24, no. 2, p. 1–10, May 1995. [Online]. Available: <https://doi.org/10.1145/568271.223785> [Cited on page 19.]
- [33] G. S. Ramachandran, K.-L. Wright, L. Zheng, P. Navaney, M. Naveed, B. Krishnamachari, and J. Dhaliwal, “Trinity: A byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence,” in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 227–235. [Cited on pages v, ix, 21, 22, 30, 36, 38, and 41.]
- [34] Y. Zhang, X. Wang, X. He, N. Zhang, Z. Zheng, and K. Xu, “Galaxy: A scalable bft and privacy-preserving pub/sub iot data sharing framework based on blockchain,” *IEEE Internet of Things Journal*, vol. 11, no. 3, pp. 5222–5236, 2024. [Cited on pages v, ix, 22, and 23.]
- [35] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, p. 114–131, Jun. 2003. [Online]. Available: <https://doi.org/10.1145/857076.857078> [Cited on pages ix, 25, and 26.]
- [36] J. Kreps, “Kafka: a distributed messaging system for log processing,” 2011. [Online]. Available: <https://api.semanticscholar.org/CorpusID:18534081> [Cited on page 27.]

- [37] O. Marcu, A. Costan, G. Antoniu, M. Pérez, B. Nicolae, R. Tudoran, and S. Bortoli, “Kera: Scalable data ingestion for stream processing,” 07 2018, pp. 1480–1485. [Cited on pages ix and 28.]
- [38] C. Jin, M. van Dijk, M. K. Reiter, and H. Zhang, “Pwop: Intrusion-tolerant and privacy-preserving sensor fusion,” *IACR Cryptology ePrint Archive*, vol. 2018, p. 1171, 2018. [Online]. Available: <https://dblp.uni-trier.de/db/journals/iacr/iacr2018.html#JinDRZ18> [Cited on pages 36 and 37.]
- [39] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, “Separating agreement from execution for byzantine fault tolerant services,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03. New York, NY, USA: Association for Computing Machinery, 2003, p. 253–267. [Online]. Available: <https://doi.org/10.1145/945445.945470> [Cited on page 38.]
- [40] I. Docker, “Docker,” <https://www.docker.com>, 2023, version 24.0 [Software]. [Cited on page 53.]
- [41] G. Combs and the Wireshark Team, “Wireshark,” <https://www.wireshark.org>, 2023, version 4.0 [Software]. [Cited on page 53.]
- [42] G. S. Veronese, M. Correia, L. C. Lung, A. N. Bessani, and P. Veríssimo, “Minimal byzantine fault tolerance: Algorithm and evaluation,” University of Lisbon, Lasige, Tech. Rep. FC–DI–Lasige 09-15, 2009. [Online]. Available: <http://hdl.handle.net/10451/14260> [Cited on page 66.]
- [43] M. Correia, N. Neves, and P. Verissimo, “How to tolerate half less one byzantine nodes in practical distributed systems,” in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, 2004, pp. 174–183. [Cited on page 66.]