TÉCNICO LISBOA

ByzPlug: Reliability Testing of BFT Systems

Ricardo Manuel Almeida Rocha

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. Daniel Simões Lopes
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Paolo Romano

October 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to start by thanking my parents for bringing me out to this world and helping me be the person I am today. Also, a big thank you to my grandparents and siblings who have given me much love throughout this journey that is life. It is thanks to all these people that I can spend time writing this investigation.

I would also like to acknowledge my dissertation supervisor Prof. Miguel Matos and Sebastião Amaro for always being present during this research, giving insights, support and sharing of knowledge that has made this Thesis possible.

A special acknowledgement to my friends João Raposo, Sidnei Teixeira and Afonso Bernardo who I have worked with throughout the last few years. I believe that working with these friends was helpful to my intellectual development throughout these years. I want to thank them for all of the knowledge we shared and the endless grinds we have been through to get things properly done on time.

Last but not least, I would like to acknowledge Francisco Pereira, João Paulo, Tânia Esteves, Maria Ramos and Hugo Rita for our discussion on some topics of this research.

To each and every one of you – Thank you.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT), using national funds as part of the projects INESC-ID UIDB/50021/2020, Ainur (financed by the OE with ref. PTDC/CCI-COM/4485/2021) and Composable's bilateral project ScalableCosmosConsensus.

Abstract

Many systems rely on Byzantine fault-tolerant (BFT) protocols for decentralized decision-making or data replication. In these protocols, participants or external systems interacting with them may behave arbitrarily, causing deviations from the intended behaviour. Moreover, translating a protocol specification into a working implementation is a complex task, which introduces challenges such as concurrency. Specifications are often oblivious to these details, leading to a substantial gap between theoretical design and practical execution, resulting in bugs. Therefore, formally proving the correctness of a protocol's specification is insufficient to ensure its reliability in real-world scenarios. Traditional testing methods, like unit testing, also fail to capture the wide range of arbitrary behaviours that can occur. To reliably assess the robustness of BFT protocol implementations, fault injection tools are essential to simulate possible byzantine behaviours. This work introduces ByzPlug, a protocol-agnostic fault injection tool that tests BFT protocols by intercepting and manipulating process-level communication over UDP/IP and TCP/IP without requiring source code extensions or modifications. ByzPlug enables message drops, replays, delays, and modifications based on protocol-specific information. We tested ByzPlug on an implementation of the HotStuff SMR protocol, exposing a safety violation, and evaluated its performance overhead on metrics like CPU usage, throughput, and latency. Results demonstrate that ByzPlug provides full observability over distributed system communication, incurring in a moderate overhead.

Keywords

Consensus, Byzantine Fault-Tolerance, Chaos Engineering, Fault Injection, Reproducibility.

Resumo

Muitos sistemas dependem de protocolos tolerantes a falhas bizantinas (BFT) para a tomada de decisões descentralizadas ou replicação de dados. Nesses protocolos, os participantes ou sistemas externos que interagem com eles podem se comportar de maneira arbitrária, causando desvios no comportamento pretendido. Além disso, traduzir a especificação de um protocolo para uma implementação funcional é uma tarefa complexa, que introduz desafios como a concorrência. As especificações frequentemente não têm esses detalhes em conta, criando uma lacuna substancial entre o design teórico e a execução prática, resultando em bugs. Portanto, provar formalmente que a especificação de um protocolo está correta é insuficiente para garantir sua confiabilidade em cenários do mundo real. Métodos tradicionais de teste, como testes unitários, também falham em capturar a vasta gama de comportamentos arbitrários. Para avaliar de forma confiável a robustez das implementações de protocolos BFT, ferramentas de injeção de falhas são essenciais para simular possíveis comportamentos bizantinos. Este trabalho apresenta ByzPlug, uma ferramenta de injeção de falhas agnóstica ao protocolo, que testa protocolos BFT interceptando e manipulando a comunicação ao nível dos processos, sobre UDP/IP e TCP/IP, sem a necessidade de extender ou modificar o código-fonte. ByzPlug permite simular a perda, repetição, atraso e modificação de mensagens com base em informações específicas do protocolo. Testamos o ByzPlug numa implementação do protocolo HotStuff SMR, expondo uma violação de segurança, e avaliamos a sua sobrecarga de desempenho em métricas como uso de CPU, taxa de transferência e latência. Os resultados demonstram que o ByzPlug fornece total observabilidade sobre a comunicação em sistemas distribuídos, incorrendo em uma sobrecarga moderada.

Palavras Chave

Consenso, Tolerância a falhas bizantinas, Chaos Engineering, Injeção de falhas, Reproducibilidade.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

QC - Quorum Commit: A set of signed messages from different replicas of a consensus protocol. Represents those replicas votes to a certain proposal in a protocol round.

1

Introduction

Contents

In a distributed system multiple computers work together to achieve a common goal. These appear to its users as a single entity, despite sometimes being dispersed across different geographic locations. Distributed systems are crucial to efficiently process large volumes of data and provide robust, fault-tolerant and highly available services. These are present in our day-to-day lives without most of us even noticing. Its use cases span from web services to distributed databases, cloud computing platforms or electronic components that make up an airplane or a spaceship.

Distributed systems require coordination and synchronisation mechanisms between all of their components. These have to agree on the same set of values throughout an execution, achieving a consensus on each of their decisions. This is a fundamental problem in distributed systems, referred to as State

Machine Replication (SMR) [2, 3] problem.

A consensus protocol is a fundamental building block of State Machine Replication and consequently of many distributed systems. It ensures that all correct processes eventually agree on a single proposed value, despite possible process failures, inconsistencies, or network failures. A working implementation of a consensus protocol relies on ensuring its Safety and Liveness properties. Cachin et al. [4] specify these properties in more detail for consensus protocols as:

1. **Agreement** (Safety): No two correct processes decide differently.
2. **Validity** (Safety): If a correct process decides v, then v was proposed by some process.
3. **Integrity** (Safety): No correct process decides twice.
4. **Termination** (Liveness): Every correct process eventually decides some value.

Some consensus protocols have fault-tolerance mechanisms implemented that only enable them to handle crash failures. Others can handle failures of arbitrary and unpredictable nature, which might be caused by nodes trying to disrupt the correct system's behaviour. Due to the famous byzantine generals problem formulated by Lamport et al. [5], these are also called byzantine nodes, and protocols that can handle the existence of these nodes are called byzantine fault-tolerant.

The Byzantine Generals Problem [5] serves as an analogy for understanding the challenges in achieving consensus in distributed systems, in which some components of this system are unreliable. It consists of a group of generals who must agree on a unified battle strategy, but among some, there might be traitors, intending to sabotage the consensus. The problem relies on the difficulty of reaching a reliable agreement when communication among generals might be untrustworthy. One general can communicate different messages to different generals, or not communicate at all.

In the context of distributed systems, this translates to the challenge of ensuring that all honest processes eventually agree on a single and consistent state, even if some processes are byzantine. More specifically, these byzantine processes might fail, provide conflicting information, decide to omit information or have another kind of arbitrary behavior that could invalidate any of the consensus properties. Moreover, even the leader can be a byzantine node and act arbitrarily.

Due to the existence of byzantine nodes in the protocol (including the leader), a slight modification in the Validity property stated for regular consensus is required. This has now to account for the possibility of a byzantine leader proposing an arbitrary value. This value can be decided by correct replicas depending on the notion of Validity considered. Cachin et al. [4] reformulate the validity property for the case of byzantine consensus to a Weak Validity and Strong Validity variants.

- **Validity** (Safety): If a correct process decides v, then v was proposed by some process.

- **Weak Validity** (Safety): If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.

- **Strong Validity** (Safety): If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\square$.

The consensus primitives must guarantee that the values decided were proposed by some process. But none of these primitives can require anything from byzantine processes. Therefore, these only restrict the actions of correct processes, and guarantee that all correct processes decide the same value despite the existence of byzantine processes.

Weak Validity allows correct processes to decide on the value proposed by a byzantine process, while Strong Validity requires that a value decided by correct processes is proposed by a correct process. Otherwise, a special value $\square$ can be decided, indicating that no valid decision was found.

Byzantine Fault Tolerance is a fundamental concept for systems in which decision-making is decentralized and any component can become a source of failure or malicious activity. Blockchains or Distributed Ledger Technologies are systems that gave relevance to many of these BFT protocols.

In a blockchain, each node is responsible for proposing and validating new blocks of transactions. Each node validates and runs these transactions before appending the new block to its chain of blocks. There has to be a consensus on the next block to be appended, which always takes into account the previous blocks of the chain. The consensus decisions determine the global state of the system.

Some of these blockchain networks have very high monetary incentives for any malicious actor wanting to diverge the consensus of the global state of the system to a state that favours its interests, such as having blocks with fraudulent transactions that highly increase the balance of its account. A well-functioning BFT protocol is what makes it harder or theoretically impossible for such a malicious actor to perform these actions.

DiemBFT [6] is a blockchain network that relies on the HotStuff [7] BFT SMR protocol. This protocol has other variants such as the Sync-HotStuff [8] or the Fast-HotStuff [9]. Cosmos [10] is a blockchain network relying on the Tendermint [11] BFT SMR protocol. HotStuff [7] improves Tendermint [11] by solving a responsiveness problem. Tendermint [11] is inspired by the famous PBFT protocol [12], which forms the basis for many other byzantine fault-tolerant protocols.

Blockchains are built on top of BFT SMR protocols, which are built on top of consensus protocols. The consensus protocols are the building block of SMR that ensures that no invalid decisions are committed by the correct blockchain nodes, consequently ensuring a valid global state of the system.

1.1 The Problem

Despite the existence of a clear and concise specification of a consensus protocol, implementing it introduces many new complex challenges. Moreover, it is not trivial to ensure the validity of consensus properties in case of multiple instances of consensus, such as in SMR protocols. The protocol specifications are often oblivious to these challenges, resulting in a substantial gap between the specification and the implementation, usually leading to implementation bugs.

In the case of byzantine consensus, the practical execution of the protocol can sometimes diverge from its theoretical design. This can happen when dealing with adverse network conditions or unexpected arbitrary behaviours from the protocol's participants.

Therefore, to assess the reliability of a byzantine fault-tolerant protocol, it is not enough to formally prove the correctness of its specification. It is also necessary to assess the robustness of the implementation. Fault injection tools are required to assess the correctness of its implementation, as other methods such as unit testing are not enough to cover the arbitrary sources of failure on these systems. These work by injecting byzantine behaviours in different components of the system, or with which it interacts and verifying how robust the implementation is against these behaviours.

Any system component can behave in many arbitrary ways, not following the protocol. The system components can decide arbitrary values, or mislead others to decide arbitrary values. The network can become unreliable (delaying the delivery, dropping or reordering packets), or its adverse conditions can lead to an inability to make progress on consensus rounds. Storage can act byzantine by writing and reading arbitrary values, or erasing records, leading to inconsistencies. All of these arbitrary behaviours can lead to the invalidation of a property of byzantine consensus.

In this thesis, we tackle the problem of ensuring that implementations of byzantine fault-tolerant protocols are reliable. More specifically, this means assessing if their implementations are robust enough to ensure the properties of consensus, even when dealing with different faults that are injected as arbitrary behaviours.

1.2 Current approaches and limitations

Several fault injection tools aim at testing the reliability of distributed systems. These are generally not agnostic to the system being tested, due to the different requirements imposed by each distributed system. However, in the case of consensus protocols, we argue that it is possible to exploit semantic similarities to develop a protocol-agnostic fault injection tool.

Such a tool should give a user the ability to reproduce and find bugs by injecting faults and verifying if any of the consensus properties were invalidated. Furthermore, this process of finding and reproducing bugs in consensus protocols could be automated.

Each of the faults injected should have the potential to influence any of the byzantine consensus properties. This maximizes the effectiveness of injecting each fault and reduces the complexity of the testing space, contributing to a more efficient and effective automatic bug search.

Some of the existing fault injection tools that focus on testing consensus protocols implement good abstractions of faults to inject, covering a wide diversity of byzantine behaviours that occur in these protocols. However, these tools are not agnostic of the protocol to which they apply. Some require extending the source code to intercept and manipulate network traffic in every new protocol to be tested [13–15]. Others [16, 17] require modifying the protocol's source code, or testing a prototype of the original protocol. This might introduce additional bugs and unexpected behaviours, leading to an unreliable assessment of the robustness of the protocol's implementation. Finally, some [18] are limited to testing protocols in a certain runtime.

All these approaches contribute in some way to the research on fault injection in consensus protocols. However, only some contribute to solving the problem we formulate in Section 1.1. Each of these comes with its set of limitations, which we explore more in-depth in Section 3.

In this work, we propose ByzPlug, a fault injection tool that attempts to overcome the set of limitations discussed in the best way possible. During the research and development of ByzPlug, we noticed some challenges that had to be overcome, which we discuss next.

1.3 Challenges to solve the problem

We identified a few key challenges that need to be addressed to solve the problem described in Section 1.1. In this section, we give an overview and briefly discuss how we overcame them.

Challenge 1: How to abstract the different possible arbitrary behaviours into the smallest subset of faults to be injected? Every component of the system and those that interact with it can be the source of a fault, by acting arbitrarily or having bugs in its implementation. It is crucial to inject faults covering all of these cases, from the system nodes to the network and possibly the storage. The faults implemented by our tool must be as small and general as possible. These should abstract the potential arbitrary behaviours that can lead to the invalidation of any of the consensus properties, consequently leading to a bug.

Despite many errors that can manifest internally and externally in a process, it is the interaction between the different protocol nodes that can lead to the invalidation of a consensus property. Correct nodes can either stop deciding or be misled by byzantine nodes into invalid decisions through their messages. Therefore, the potential behaviours can be abstracted in two ways: whether a node is still participating in the protocol, and how it is participating in the protocol (what is the content of its messages and how it handles replayed messages). As a result of this abstraction, we implement three

faults: Message omission, message modification, and message replay.

We argue that these three faults are enough to capture most errors that occur in the protocol's code, or different byzantine behaviours that could lead to the invalidation of one or more of the consensus properties.

Challenge 2: How to build a fault injection tool that can inject the implemented faults in any protocol? Every protocol is different in some way. These are written using different programming languages, the structure of their messages differ, and different serialization libraries are used. Moreover, some protocols take advantage of TCP to ensure reliable communication between nodes, while others use UDP. Despite all of this, it is possible to have a tool able to inject faults in any protocol without requiring additional source code modifications or extensions. Achieving that requires: *i*) finding fault abstractions that are common to every protocol; *ii*) having the right tools to implement faults based on those abstractions.

These protocols run in a set of nodes exchanging messages with each other. These messages are nothing more than chunks of bytes, which can be intercepted and modified. Moreover, the transmission of those bytes to a given receiver can be stopped, delayed or replayed multiple times. These behaviours represent the abstractions. Implementing them requires tools that give the ability to perform these actions before packets get to the network, or are delivered to the receiver.

We argue that the eBPF Framework [19] and eXpress Data Path (XDP) [1] are suitable tools to overcome this challenge. In Sections 2.7 and 2.8 we give the required background to explain how these tools can help overcome this challenge.

Challenge 3: How to manipulate TCP/IP communication, given the reliable communication mechanisms implemented by TCP? Transport Control Protocol (TCP) ensures reliable bidirectional communication between two nodes. It will ensure that messages are not dropped, modified or delivered in duplicate. Some protocols take advantage of TCP to have these guarantees, instead of implementing them from scratch. To inject faults consisting of message drops, modifications and replays we need to find a way to overcome TCP's reliability mechanisms. We discuss all the details of how we achieve this challenge in Section 4.5.

Challenge 4: How can the fault-injection tool consider protocol-specific information while still working with any protocol? Reproducing bugs strategically requires the ability to access and modify protocol-specific information. It is crucial to distinguish the different types of messages exchanged, read or modify specific fields of those messages. Users of a fault injection tool should be able to achieve this using the same programming language and libraries that were used in the original protocol's source code. Meanwhile, it is important that the fault injection tool can still be used with any protocol, maintaining its protocol-agnostic feature. In Section 4.5 we describe how we solved this challenge.

Challenge 5: How to manipulate application-level data given the restrictions imposed by the eBPF verifier? In Section 2.6 we describe the restrictions imposed by the eBPF verifier, and the reason for these restrictions. Due to these, we argue that eBPF [19] is not suited for writing complex programs which parse and manipulate application-level data. Therefore, we need to look for a solution that sends the packet data to user space, where it can be freely manipulated without these restrictions before any processing is done by the network stack. In Section 4.1 we give an overview of the necessary infrastructure that our implementation relies upon to achieve this.

Challenge 6: How to drive the state of protocol nodes to a specific state in which a bug might manifest? Some bugs only manifest when the protocol nodes simultaneously reach a specific state. To achieve this, one can either: *i*) run multiple protocol executions until that specific state is present in each of the nodes; or *ii*) run a single controlled protocol execution, driving the state of each node to that required state.

The second approach enables a user to more accurately reproduce a bug. Achieving it requires enforcing a view-based communication and deterministic leader election. In each view, the user controls the nodes that propose messages, the amount and content of those proposed messages, the nodes that vote for those proposals, and the amount and content of those votes. Nevertheless, this approach is only possible if the fault injection tool considers protocol-specific information.

Challenge 7: How to tackle the challenges while keeping a low overhead? A high overhead leads to a big discrepancy between the timing in which events occur in a normal execution and an execution running alongside the fault injection tool. It is fundamental to keep a low overhead so that the fault injection tool does not compromise any of the testing results. To ensure this, the fault injection tool should have low complexity. To tackle this challenge, we implement faults as programs of low complexity using Linux kernel sub-systems such as eBPF and eXpress Data Path (XDP).

1.4 Objectives

The objective of this research is to come out with a solution that facilitates a reliable assessment of the robustness of different byzantine fault-tolerant protocol implementations. Such a solution should be able to be used with any protocol, despite the different libraries and programming languages used to develop it, without requiring any source code extensions or modifications.

1.5 Achievements

We built a protocol-agnostic fault injection tool that gives a user the ability to intercept and manipulate all UDP/IP and TCP/IP process-level communication in a network. It can be used to manually find and reproduce bugs in any implementation of a byzantine fault-tolerant protocol, without requiring any extensions or modifications to its source code.

We tested the Two-Chain and Three-Chain variants of a C++ implementation of HotStuff [20] using ByzPlug. Both variants are robust to many different byzantine behaviours tested. However, we found a safety violation in the Two-Chain variant of this HotStuff implementation. We give an in-depth explanation of how to reproduce this bug in Section 5.1.

1.6 Outline

The rest of this document is organized as follows: Section 2 introduces the necessary background on the different concepts used throughout this thesis. Section 3 presents current state-of-art solutions for fault injection. Section 4 describes ByzPlug specification and how it is implemented. Section 5 presents the result of evaluating a BFT protocol, in which a safety violation is reproduced, and ByzPlug's performance metrics. Section 6 concludes the thesis and explains future work that could be done to improve ByzPlug.

2

Background

Contents

In this chapter, we introduce the background concepts used throughout this work. We start by introducing the Linux Network Stack in Section 2.1. We go more in-depth regarding a specific layer of this network stack in Section 2.2, the Transport Layer, and discuss the Transport Control Protocol. In Section 2.3 we present some Linux Networking Concepts, such as address families, network namespaces, address families, bridges, virtual ethernet interfaces and containers. Following this, we introduce some technologies used in the development of our solution, from Section 2.4 to Section 2.7. These are

Protocol Buffers [21], gRPC [22], the eBPF Framework [19] and the eXpress Data Path (XDP) [1]. In Section 2.8 we go more in-depth about one of the address families, AF_XDP [23] and explain how sockets of that address family are crucial to overcoming some of the challenges mentioned in Section 1.3. Finally, in Section 2.9 we give background on HotStuff [7], a byzantine-fault tolerant protocol that we tested using ByzPlug.

2.1 Linux Network Stack

The Linux network stack [24] is one of the most extensive kernel subsystems. It is referred to as a "stack" due to its layered composition of protocols. Each protocol layer is responsible for specific tasks, working together to ensure the reliable delivery of data from the sender to the intended receiver without modification or loss.

The topmost layer is the application layer. In this layer, system calls such as *sendmsg()* and *recvmsg()* can be used to send or wait for the delivery of a message.

Below the application layer is the transport layer, responsible for ensuring reliable data transmission. Transport protocols, such as Transmission Control Protocol (TCP) [25] handle error checking, retransmissions, and reordering of packets. This ensures that no packets are lost or duplicated, and all packets are delivered to the application in the same order they were sent.

Next to the transport layer is the internet layer. In this layer, an Internet Protocol (IP) ensures that packets are forwarded and reassembled properly as they traverse different networks and routers to reach their destination.

Beneath the internet layer is the data link layer (also referred to as the MAC layer). It provides a bridge between the network protocols and the Network Interface Card (NIC). In this layer, packets are placed into frames and data transfer to and from the NIC is managed using ring buffers.

Finally, there is the physical layer, composed of the physical medium (cables, or wireless signals) that connects two NIC devices.

When a message is sent using the *sendmsg()* system call, it is encapsulated in a packet that is incrementally built as it moves down the stack, from the application layer to the NIC. Once it reaches the NIC, the packet is transmitted across the physical medium and eventually arrives at the receiver's NIC. The message then traverses upward through the receiver's network stack, from the physical layer to the application layer. Throughout this traversal, in each layer the respective packet's header is parsed and according to its information the layer's protocol ensures its responsibilities, such as routing, error correction, and packet integrity—are fulfilled. Figure 2.1 illustrates this path taken by a packet.

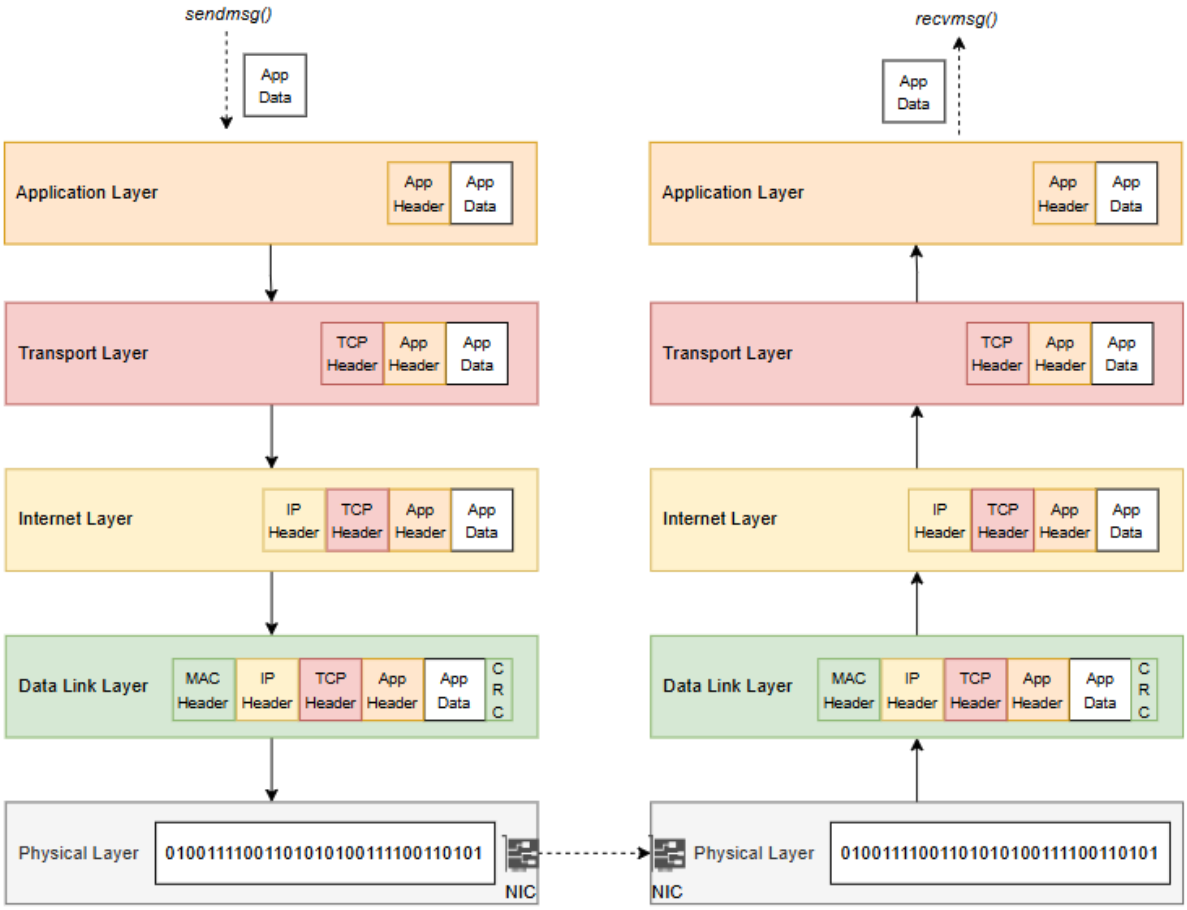

Figure 2.1: Illustration of the path of a packet through the different layers of Linux Network Stack.

2.2 Transport Control Protocol (TCP)

The Transmission Control Protocol (TCP) [25] operates at the transport layer. It is designed to provide reliable bidirectional communication between two devices. TCP ensures that no packets are lost, duplicated, or delivered out of order. This offers a level of reliability essential for many network applications, alleviating them from the burden of implementing these mechanisms.

A TCP connection begins with a process known as a 'three-way handshake', in which a connection between a client and server is established. During this handshake, the two parties agree on an initial sequence number, which serves as a reference point to ensure the reliability of the TCP connection.

Each time a message is sent, TCP includes the current sequence number in the packet header. Furthermore, this number is incremented in the sender's TCP state machine by the number of application data bytes in the transmitted packet.

At the receiver's side, before any message is delivered to the application layer, TCP checks whether the sequence number of the received packet matches the expected value. In case it does, the packet is

delivered to the application layer and TCP sends back an acknowledgement packet to the sender. This packet includes an acknowledgement number, used to inform the sender of the next expected sequence number. This number is computed by incrementing the number of received bytes to the last expected sequence number.

Figure 2.2 illustrates the three-way handshake process, in which Alice starts a bidirectional communication with Bob through TCP. From this point, Alice is expecting TCP packets from Bob with sequence number m+1. Bob is expecting TCP packets with sequence number n+1. Neither Alice nor Bob will forward any packet's data to the application layer in case the packet has a lower or higher sequence number than the expected.

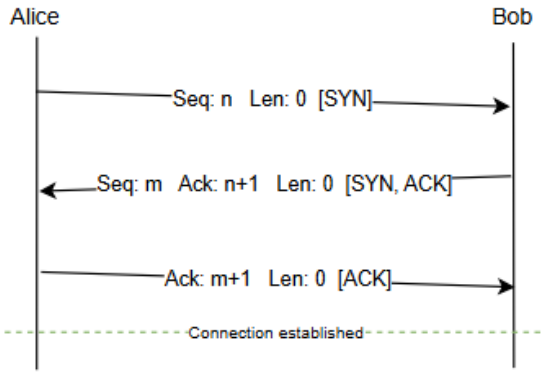

Figure 2.2: Illustration of the three-way handshake process of TCP.

2.3 Linux Networking Concepts

- **Linux Address Families**

  An address family (or communication domain) in Linux specifies the protocol family used for communication. These families are used when creating a socket to specify the type of communication desired. Examples of some address families are:

  - **AF_UNIX:** Used for local (intra-host) communication between processes on the same machine via Unix domain sockets.
  - **AF_INET:** Supports IPv4-based communication for internet or network connections.
  - **AF_INET6:** Supports IPv6-based communication, offering a larger address space for internet or network connections.
  - **AF_XDP:** Used for high-performance packet processing in user space (explained in detail in Section 2.8).

- **Linux Network Namespaces**

  A Linux network namespace is an isolated network environment within the operating system. Each network namespace has its network interfaces, IP addresses, routing tables and firewall rules. As a result, for each network namespace, there is a separate virtualized instance of a network stack. This is particularly useful in virtualized network environments, where different applications or services require isolated network settings while running on the same host machine.

- **Linux Bridges**

  A Linux bridge functions as a virtual network switch. It operates at the data link layer, examining destination MAC addresses of incoming packets and forwarding them to the appropriate interface within its network. Linux bridges are commonly used in scenarios such as routers, gateways and virtualized environments. These enable communication between virtual machines or network namespaces on a single host, without the need for additional routing.

- **Linux Containers**

  Containers are lightweight, portable units of software that package an application, its dependencies, libraries and configuration files in a self-contained environment. Unlike virtual machines, containers share the host system's kernel but maintain isolated user spaces. They have their filesystem and run on their network namespace, having their own network stack, and process space. Containers can run on a single host and consume fewer resources than virtual machines. Moreover, containers ensure that an application can run consistently regardless of the underlying infrastructure.

  Docker [26] is a platform that simplifies the creation, deployment, and management of containers. By leveraging Linux container technologies, Docker provides developers with an easy-to-use interface and tools for packaging applications and their dependencies into containers. Docker also includes networking capabilities that allow containers to communicate with each other and external systems through features like Docker networks, virtual ethernet (veth) interfaces, and Docker bridges.

- **Virtual Ethernet Interfaces**

  Virtual Ethernet Interfaces are a type of Linux interface commonly used in virtualized environments. These are created in pairs, and function as a virtual ethernet cable connecting two endpoints. Packets transmitted on one end of the pair are immediately received on the other end. This creates a bidirectional communication channel between both ends of the `veth` pair.

  Virtual Ethernet interfaces serve as a bridge between isolated network environments, such as network namespaces and the host or external networks. When a container is deployed, a virtual

ethernet interface is assigned to it. One of its ends (`eth0`) is connected to that container's network namespace, while the other end (`veth`) is connected to the host namespace. In case there is a bridge connecting both containers, the `veth` end is connected to the bridge in the host namespace. Figure 2.3 illustrates this case.

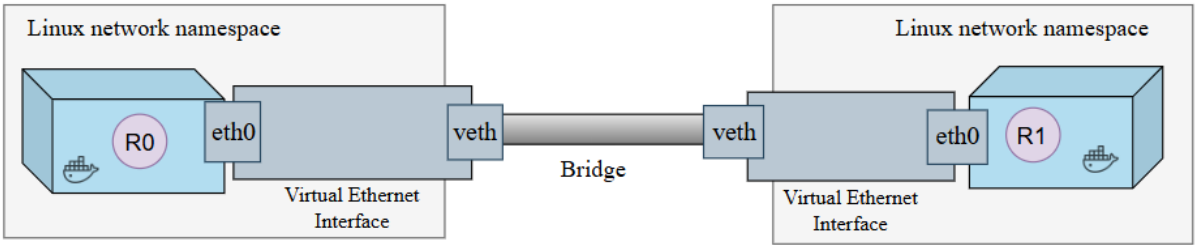

Figure 2.3: Two containers connected to the same bridge through their virtual ethernet interfaces. The `eth0` virtual device is in each container's network namespace. The `veth` virtual device is in the same namespace as the bridge.

2.4 Protocol Buffers

Protocol Buffers (Protobufs) [21] are a language-agnostic, platform-neutral mechanism developed by Google for serializing structured data. Protobufs enable the definition of data structures (messages) using a simple interface definition language (IDL), which is then compiled into source code in various programming languages. This source code provides both serialization and deserialization functionality, allowing structured data to be efficiently transmitted between systems.

2.5 gRPC

gRPC [22] is a high-performance, open-source remote procedure call (RPC) framework developed by Google, built on top of HTTP/2 and designed to work seamlessly with Protocol Buffers (Protobufs) for data serialization. gRPC enables efficient communication between distributed services and clients by allowing functions or methods to be called remotely as if they were local while abstracting the underlying network communication.

Using Protobufs as its interface definition language (IDL), gRPC generates both client and server code, enabling communication between services written in different programming languages. gRPC provides a robust mechanism for defining service contracts through Protobufs, where services and their methods are specified.

2.6 eBPF Framework

In the field of systems programming and performance monitoring, eBPF (extended Berkeley Packet Filter) [19] has emerged as a revolutionary technology. It was initially conceived for network packet filtering but has evolved far beyond its original design. This powerful framework now allows for dynamic analysis, monitoring, and modification of the behaviour of a system in real time.

It works by dynamically instrumenting the binary code running in both the kernel and user space. This allows users to write eBPF programs in languages such as C, which can be hooked to specific points of the kernel and programs in user space. These intercept every event occurring in the hook point to which they are attached, and allow for collecting information or modifying its behaviour. Key hook points for eBPF programs consist of:

- **Linux Traffic Control layer and the Network Interface Card driver:** Monitoring these components of the network stack gives us the ability to observe and manipulate all process-level communication in a network.
- **System calls:** Monitoring both the invocation and return of system calls not only provides us with a detailed view of the interactions between user space and the kernel but also enables us to have full control over these.
- **Function entries and exits:** Monitoring both entries and exits of functions provides insights about their invocations and return values, giving us full control over these.

It should be highlighted that the first two hook points are exclusive to kernel space. Nevertheless, the third can be user space defined functions or kernel functions.

eBPF programs execute in response to specific triggers at their attached hook points. Since eBPF programs do not have access to persistent storage within their context, Linux provides a key-value store known as eBPF map to persist data.

eBPF maps are defined at program load time. These enable data sharing between the kernel and user space, as well as between different instances of eBPF programs. The existence of eBPF maps enables persistent data storage between invocations of the same, or different eBPF programs.

Since eBPF code runs directly in the kernel address space, it can directly access, and potentially corrupt kernel memory. To prevent this from happening, the kernel enforces a single entry point for loading all eBPF programs (through the *bpf()* system call).

Before a program is loaded into kernel space it is first analyzed by the in-kernel eBPF verifier. The verifier performs a static analysis of the program byte code to ensure that the program performs no unsafe actions (such as accessing arbitrary memory) and that the program will terminate.

This implies that standard C libraries can not be used and instead, the user must either rely on an existing set of functions that the verifier accepts, called *bpf* helpers or write the equivalent functions from scratch with the necessary safeguards included.

The existence of a verifier is fundamental for deploying safe code to the kernel. However, this strict verification process presents significant challenges, particularly when writing complex programs.

2.7 eXpress Data Path (XDP)

The eXpress Data Path (XDP) [1] is a framework for programmable packet parsing that is part of the Linux kernel. XDP enables custom processing (including redirection) at the earliest possible point after a packet is received from the hardware, and before the kernel has access to its data.

Programmable packet processing is increasingly implemented using kernel bypass techniques (such as DPDK [27]), where a userspace application takes complete control of the networking hardware to avoid expensive context switches between kernel and userspace. However, as the operating system is bypassed, so are its application isolation and security mechanisms.

XDP [1] takes an approach that is the opposite of kernel bypass: Instead of moving control of the networking hardware out of the kernel, the performance-sensitive packet processing operations are moved into the kernel, and executed before the operating system networking stack begins its processing.

This retains the advantage of removing the kernel-userspace boundary between networking hardware and packet processing code, while keeping the kernel in control of the hardware, thus preserving the management interface and the security guarantees offered by the operating system.

While XDP allows packet processing to move into the operating system for maximum performance, it also allows the programs loaded into the kernel to selectively redirect packets to a special user-space socket type (AF_XDP), which bypasses the normal networking stack.

There is an important detail that must be highlighted: XDP only intercepts packets at the RX interface of the NIC. It can not intercept packets at the TX interface, meaning it only takes action on packets arriving at the NIC.

When a packet is intercepted by an XDP program the following actions can be returned:

i) **XDP_PASS:** the packet is processed by the network stack.
ii) **XDP_DROP:** the packet is dropped.
iii) **XDP_REDIRECT:** the packet is redirected to another network device.
iv) **XDP_TX:** the packet is bounced out of the same network device it arrived on.

Figure 2.4 illustrates the XDP integration with the Linux network stack and the data flow of a packet when each of these different actions is returned by an XDP program. Apart from this, it illustrates the

interaction between XDP programs and eBPF maps, so that information can be shared between kernel and user space.

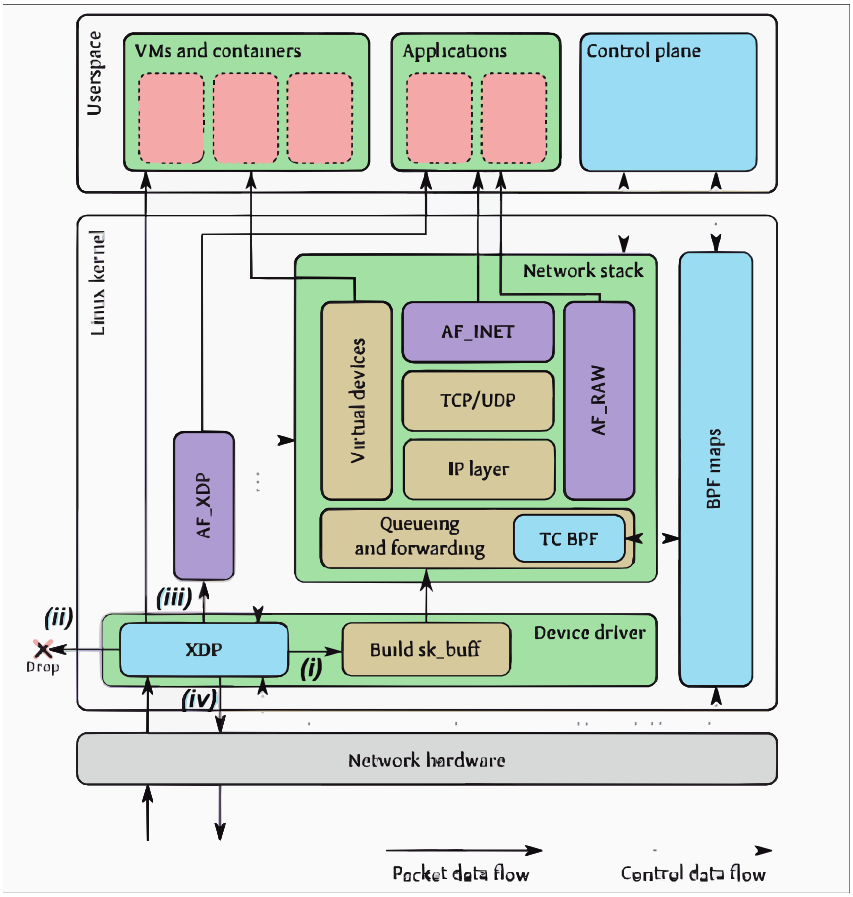

Figure 2.4: XDP integration with Linux network stack. Figure from the official XDP research paper [1].

2.8 AF_XDP Address Family Sockets

AF_XDP [23] is an address family that is optimized for high-performance packet processing. Sockets from this address family are called XSK. These enable the possibility for XDP programs to redirect frames to a memory buffer (UMEM) managed by a user-space application associated with that socket.

Two rings are associated with each AF_XDP socket: RX ring, where the socket can receive packets; and TX ring, from which the socket can send packets. An entry in one of these rings is filled with a descriptor that consists of the offset of a packet placed in the UMEM. Although each socket has its UMEM, RX and TX rings, these can also be shared between different sockets.

Packets can be redirected from an XDP program to one of these sockets (XSK) by redirecting the packet to a special *bpf* map referenced as XSKMAP. The user-space application can place an XSK de-

scriptor at an arbitrary place in this map, and the XDP program can then redirect a packet to that specific index in this map. At this point, the packet's frame is placed in the XSK's UMEM and its offset in the UMEM is placed in a descriptor in the socket's RX ring.

AF_XDP [23] is an address family that is optimized for high-performance packet processing. These allow XDP programs to redirect packet frames to memory buffer in user space (UMEM), bypassing the kernel network stack. Although each socket has its UMEM and two rings (RX and TX), these can also be shared between different sockets.

An XSK continuously attempts to retrieve redirected frames from the UMEM by continuously invoking *poll()*. When accessible, the frame's data can be manipulated by a user space C or C++ program, instead of a kernel space eBPF program.

An important detail should be highlighted: after a frame is redirected to user space there is no support to inject it back to the kernel network stack. The only way to do this is by receiving the packet in the same interface it was originally intercepted in.

Figure 2.5 illustrates the sequence of steps taken by the user space programs associated to these sockets: *1*) the packet's frame is redirected by an XDP program to a special map (XSKMAP) associated with that XSK. Consequently, the frame is placed in that socket's UMEM and a descriptor pointing to the offset of the frame in the UMEM is placed in the socket's RX ring; *2*) *poll()* returns the frame descriptor in the RX ring and the user space program can fetch the frame from the UMEM and manipulate it, possibly injecting faults. After this, the descriptor can be placed in the TX ring; *3*) the *sendmsg()* system call is invoked, so that the frame in the TX ring is sent out of the socket; *4*) the descriptor removed from the TX ring.

2.9 HotStuff

HotStuff [7] is a Byzantine Fault Tolerant protocol that solves the State Machine Replication (SMR) problem. At the core of SMR is a protocol for deciding on a growing log of *command* requests by a client. A group of protocol nodes apply commands in sequence order consistently.

The Basic HotStuff [7] consensus protocol works in a sequence of *views* numbered with monotonically increasing view numbers. Each *viewNumber* has a uniquely dedicated leader known to all. Each replica stores a tree of pending commands as its local data structure. During the protocol, a monotonically growing branch becomes committed. To become committed, the leader of a particular view proposing the branch must collect votes from a quorum of (n-f)[1] replica in each of the three protocol phases: Prepare, Pre-Commit and Commit. This collection of signed votes from (n - f) replicas, referred to as *quorum certificate* (or "QC" in short) is a key step of the protocol. Each QC is associated with a

[1] n is the total number of replicas, while f is the maximum number of byzantine replicas tolerated in the system

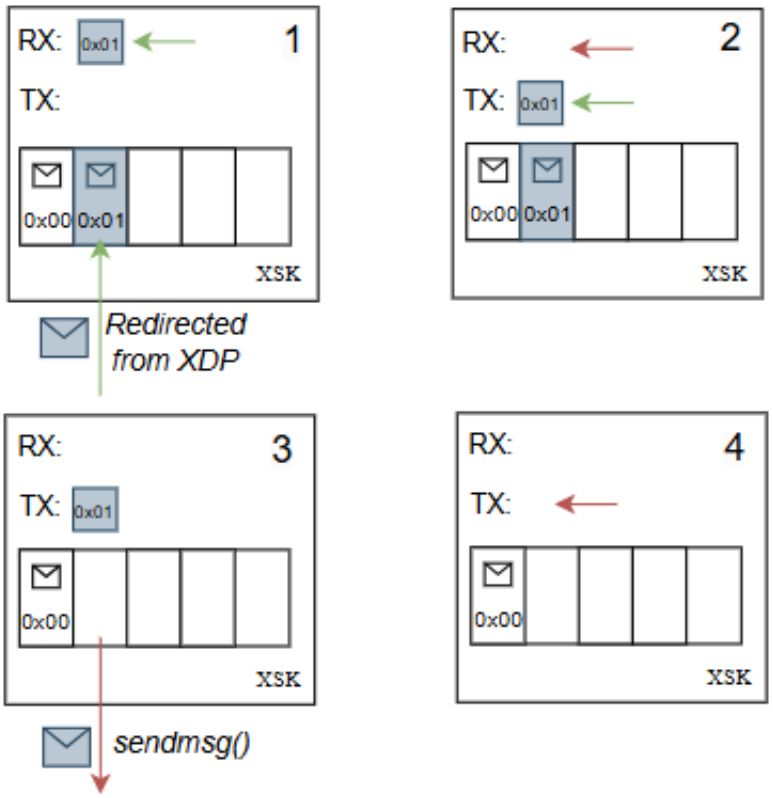

Figure 2.5: Sequence of steps taken by a XSK user space program

particular node (that collects the votes) and a view number.

The three protocol phases do not do anything other than collect votes from replicas, and they are very similar. Chained HotStuff [7], an improved version of Basic HotStuff optimizes this, reducing the number of messages and allowing for pipelining of decisions. The idea is to change the view on every Prepare phase so that each proposal has its own view.

More specifically, in Chained HotStuff the votes over a Prepare phase are collected in a view by the leader into a QC. Then, the QC is relayed to the leader of the next view, essentially delegating responsibility for the next phase, which would have been the Pre-Commit to the next leader. However, the next leader does not carry with a Pre-Commit phase, but instead initiates a new Prepare phase, adding its proposal. As a result, in Chained HotStuff a Prepare phase for view v+2 simultaneously serves as a Pre-Commit phase for view v+1 and as a Commit phase for view v. This is possible because the protocol messages of each of these phases have the same structure.

There are two variants of Chained HotStuff: Two-Chain and Three-Chain. To understand the difference between these we first need to introduce the concepts of One-Chain, Two-Chain and Three-Chain.

A One-Chain is formed when a block has a QC pointing to its direct parent. A Two-Chain is formed when a block has a QC pointing to its direct parent and that parent forms a One-Chain. A Three-Chain

is formed when a block has a QC pointing to its direct parent and that parent forms a Two-Chain.

The Two-Chain version of Chained HotStuff only requires a Two-Chain for a node to commit a block, and a One-Chain for a node to lock on a given block. Similarly, the Three-Chain version requires a Three-Chain for a node to commit a decision, and a Two-Chain to lock on a given block.

While the Three-Chain degrades throughput in cases in which many view changes occur, the Two-Chain HotStuff variant loses Optimistic Responsiveness, consequently requiring a separate module referred to as Pacemaker to ensure the protocol's liveness. The Pacemaker is responsible for triggering view changes and controlling leader election for each view.

3

Related Work

Contents

In this chapter, we present several fault injection tools. Some are presented to discuss different approaches for fault injection (Section 3.1 and Section 3.2). Those presented in Section 3.3 focus on testing the implementation of consensus protocols and are presented to compare them to the solution we develop to solve the problem formulated in Section 1.1. In Section 3.4 we compare each of these tools using a set of criteria that help expose their sets of best features and limitations.

3.1 Fault Injection for testing Distributed Systems

Each distributed system has a specific use case and thus requires a different degree of fault tolerance. In some distributed systems a simple process crash at a crucial moment can either lead to bugs that make the system hang forever or an impossibility to recover the fault of the crashed process. Bugs

that fit in this category are called Time of Fault (TOF) bugs, given that these only occur in very specific moments of execution.

Time of Fault bugs [28] manifest when a node crashes in the middle of a state update routine after writing some information required to start the routine, to heap or persistent storage, making it impossible for another thread to finish the update routine and effectively recover the fault of the crashed node.

Other distributed systems, such as Cluster Management Controllers [29] leave no margin for errors in state updates. In these, the end state must be the exact specified desired state for the system. Otherwise, the controller does not serve its purpose and has no use case.

Liu et al. present FCatch [28] to detect TOF bugs, by modeling them as a new kind of concurrency bugs and separating them into two different types: Crash-Regular and Crash-Recovery bugs, both modeled as pairs of conflicting Read and Write operations that are related through Causal and Blocking relationships, which are the key pieces for establishing the bug candidates. These are found through an exhaustive search through the huge amount of different traced operations.

The trace is obtained by instrumenting the heap and persistent data storage operations. This leads to many candidates of conflicting operations having to be pruned by a costly static and dynamic code analysis that looks for data and control dependencies. This analysis is done using the WALA framework [30] and the instrumentation of the heap is done through Javassist [31] (Java bytecode instrumentation), thus limiting FCatch usability to testing applications in JVM runtimes.

Despite selectively instrumenting the heap, the overhead of tracing all the operations, looking for the relationships among them, and invalidating them with this code analysis imposes a huge overhead. Although having this big downside, FCatch can be run without any user input and can automatically generate its test cases for injecting faults. It also outputs bug reports with the respective trace containing the operations that should be reproduced to trigger the bug.

Sun et al. present Sieve [32] to detect bugs in Cluster Management Controllers, such as Kubernetes [29] (for which it was specifically tested). These managed clusters are distributed systems following a state-reconciliation pattern, which is based on having a controller constantly driving their current state to a desired one. The controller exchanges sequences of Notifications and Updates with the cluster through state-centric interfaces.

State-centric interfaces consist of an abstraction implementing read, write, and notification operations. By introspecting these interfaces, both controller traces and cluster state traces can be collected. The former is used to automatically generate the test plans, which are used to specify the moment in which faults will be injected into the system. The latter is used to test the safety and liveness properties of the system after injecting the faults.

The test plans generated attempt to trigger TOF bugs (also called Intermediate-State in Sieve) and

other bugs as a consequence of repeated or missing state updates. These are generated exhaustively, leading to further pruning. The pruning decision is based on the existence of relationships between Notification and Update operations contained in these cases. In case any bugs are reported, their respective test cases are saved so that the execution trace leading to those can be reproduced.

To test another controller, the user needs to instrument a few methods of that specific controller. This methodology can also be used to test other systems that are not controllers as long as they rely on state-centric interfaces.

3.2 Chaos Engineering

Chaos Engineering is an interesting approach to fault injection in which the behavior of a system is evaluated in a production environment. First, some metrics are collected in runs in which faults are not injected into the system. Then, the same set of metrics are collected in runs in which faults are injected into the system. In the end, metrics are compared to assess how each is affected when some faults are injected into the system.

The moments in which to inject faults are automatically generated based on stochastic models. Despite the big advantage of not having to specify these moments, due to the probabilistic nature of the faults, it is hard to reproduce bugs that manifest.

Zhang et al. present Phoebe [33] which monitors system call invocations, together with some application-specific metrics of the target system. Each system call invocation is associated with success or error, and its respective return value, which contains the error code in case of an error.

For each different error code returned by a system call invocation a stochastic error model is computed, amplifying realistically that error rate. The point is to assess whether errors that are not harmful to the system when occurring at low rates might be harmful when occurring at realistic higher rates. Faults are injected by changing the return value of some successful system calls to the error code to be injected.

The monitoring and modification of return values of system calls is achieved through the use of the eBPF framework [19]. The application-specific metrics are collected by instrumenting the Java bytecode. Despite the approach taken to monitor and modify the behavior of system calls, which is extensible to any system and runtime, the approach taken to monitor the application-specific metrics limits the usability of this tool for testing applications in JVM runtimes.

Zhang et al. also present ChaosETH [34] which is similar to Phoebe [33], following the same monitoring and fault injection approach. The difference, in this case, is that the applications being tested are Ethereum blockchain clients (namely GoEthereum [35] and Nethermind [36]), so the chosen

metrics are specific to each client. The component of ChaosETH [34] responsible for collecting the metrics is general for any blockchain client exposing a public API that can be queried. To extend it to test other Ethereum blockchain clients, a user-configured file with client-specific metrics URLs should be provided.

Sondhi et al. present a fault injection tool based on Chaos Engineering to evaluate the performance of consensus algorithms that run on permissioned blockchains [37] (PBFT [12], Clique [38], and Raft [39]). In this case, the metrics chosen are transaction throughput (transactions per second), transaction latency (time between the creation and the commitment of a transaction), and appended block success rate (ratio between appended and created blocks).

A different approach for fault injection is taken given the focus on testing consensus protocols. These are tested in a blockchain testbed where platform-specific applications are deployed. Testing another protocol requires deploying an application to a testbed in the blockchain in which the protocol is running.

The protocols being tested are subject to different workloads. These are injected with the help of the Locust framework [40], which records the time for the response of each HTTP request sent to the system. Using the Pumba framework [41] process faults such as crashes and message corruptions, and network faults such as message drops and delays are injected stochastically.

3.3 Fault Injection for testing Consensus Protocols

As discussed in the Introduction, fault injection is fundamental to assessing the reliability of consensus protocols, mainly those that are byzantine fault tolerant. Their implementations should be robust to arbitrary behaviors, ranging from adverse network conditions to byzantine behaviors from internal and external system components. There are two kinds of approaches to testing these protocols: one that tests a simulation of the protocol and another that tests its real implementation.

3.3.1 Testing a simulation of a protocol

Some fault injection tools provide a simulator that enables users to quickly prototype a protocol or parts of it and test its performance, correctness, or both.

Wang et al. [17] present a flexible simulator for byzantine fault tolerant protocols. The user has to implement all the necessary modules for the simulator to work. It evaluates the protocol's performance under different network conditions and process faults, requiring the user to specify the moments in which these faults are injected. The performance metrics considered are *i*) time to complete one round; and *ii*) number of messages exchanged during one round.

All the messages exchanged between the nodes are saved and reported in an execution trace at the end of testing. This can be used to compare with execution traces produced by the original version of the algorithm to test the correctness of the protocol's prototype.

Gupta et al. present BFT-Bench [16] to test the performance of BFT protocols. It uses as performance metrics *i*) transaction latency; and *ii*) transaction throughput. It requires the user to specify the moments in which faults are injected, and has a good degree of fault coverage, by supporting the injection of faults such as system overloading, replica crash, message delay, and message flood.

It supports six different protocols. In case a user wishes to integrate a new one, the requirements are to implement the message delay and message flood behaviors specific to the protocol to be integrated, and prototyping the client emulator for the workload injector.

3.3.2 Testing the real implementations of a protocol

Soares et al. present Zermia [18], a fault injection framework for testing BFT protocols. It is based on a Client-Server architecture, in which a coordinator distributes user-defined fault schedules to agents. Agents are separate threads that share the runtime with the main processes of the application being tested. Their task is to trigger the faults defined in the schedules when conditions are met. These conditions are not solely based on time but also dependent on other events such as the occurrence, or the absence of certain faults. Due to this, faults can be triggered deterministically.

Five different predefined faults can be triggered, namely: delay thread, process crash, message omission, message drop, and message flood. On top of this existing diversity of faults, the user can define other customized faults by implementing some methods of an interface. Faults are injected using AspectJ [42], a Java bytecode instrumentation method.

This is a highly flexible tool, incurring a minimal overhead given that faults are injected in compile time and triggered in a separate thread at runtime. Still, it has two very big downsides: it can only test applications in JVM runtimes, and the user has to specify the fault schedules. The latter makes it harder to find new bugs in a protocol, given that it requires the user to reason about the combination of events that could lead a bug to manifest.

Drăgoi et al. [14] exploit the semantic properties of consensus to reduce an execution trace of higher complexity and harder interpretability to another one in which the same set of bugs manifest but having lower complexity and easier interpretability.

This is achieved by taking advantage of the Communication-Closure property of consensus algorithms, and other techniques such as: *i*) specifying the moments in which faults are injected using protocol rounds; *ii*) bounding the number of rounds in which process faults can occur; and *iii*) bounding the time (in rounds) that the processes take to recover.

Test cases are generated by an algorithm, which chooses at random the moments in which faults are injected, according to the techniques above. These faults consist of process crashes and message drops.

Any bug manifested due to the invalidation of a consensus property is reported together with the sequence of events leading to it. Testing a new protocol requires extending its source code so that faults can be injected and properties verified.

The main contribution of this work is showing how we can exploit the semantic properties of consensus to reduce the huge space of possible executions to a smaller space in which the same amount of bugs manifest.

Winter et al. [15] take advantage of these previously mentioned techniques by Drăgoi et al. [14] and applies it to the testing byzantine fault tolerant protocols. The authors further present other techniques to reduce the number of different executions to sample from.

These techniques are based on mutating the messages in certain ways that are more likely to interfere with the current protocol's execution. Instead of making arbitrary mutations to a message, only specific fields are changed. Examples are small increments or decrements of the round number in a message, and reusing values previously committed on successful phases of the protocol.

Apart from mutating messages, crashing processes and dropping messages are also faulty behaviours that this tool provides to test the reliability of a byzantine fault-tolerant protocol's implementation. To inject the faults specified in the generated test case, an interception layer is implemented specifically for each new protocol to be tested.

Bano et al. [13] introduce Twins, a fault injection tool that *i*) generates test cases containing byzantine behaviours such as leader equivocation, incoherent voting, double voting and message drops; and *ii*) executes instances of the protocol in which these faults are injected according to the generated test cases, further verifying if any of the byzantine consensus properties were invalidated.

A component called Scenario Generator is responsible for *(i)* generating the different test cases. Each of these is represented by a different combination of the compromised nodes and per-round network partitions and leaders. With this approach, all the possible message delivery schedules and leader combinations for each round are enumerated. Following this exhaustive generation of test cases, a pruning phase is required to remove those that are irrelevant or duplicate.

A component called Scenario Executor is responsible for *(ii)* executing an instance of the protocol according to a generated test case. It starts by deploying the network configuration specified in the test case in an emulated network. The emulated network can be deployed using a module implemented in the DiemBFT open-source repository referred to as Network Playground [43]. It consists of a multiplexing wrapper over the system to be tested, providing the ability to intercept all the messages exchanged

between the nodes of the system and inject faults as specified in the test cases.

To inject faults, it instantiates a *Twin* in the Network playground - an exact copy of one of the nodes in the network. It has the same credentials and identities, being indistinguishable from its original replica, but acting arbitrarily according to the faulty behaviours specified in the test case being executed.

3.4 Discussion

The fault injection tools discussed in Section 3.1 [28, 32] are either too restricted to a certain system's use case [32] or consider a subset of faults that is too small to cover the diversity of byzantine behaviors that can occur in a byzantine fault-tolerant protocol.

Similarly, most of the tools discussed in Section 3.2 [33, 34] do not consider a wide enough subset of faults to be effective in testing the reliability of byzantine fault-tolerant protocols. There is an exception, but its focus is on testing the performance of consensus protocols instead of the correctness [37], not aligning with our main objective.

The faults covered by these tools [28, 32–34] are more likely to result in behaviours such as process crashes, message corruption, incoherent voting, double voting, or leader equivocations. As we argued in Challenge 1, reproducing or finding bugs should be done by directly injecting behaviours that interfere with the consensus properties, instead of having them as a consequence of other injected faults.

The tools presented in Section 3.3.1 [16, 17] are very flexible, letting the user customise the faulty behaviours. However, these are based on simulation, not testing the real protocol implementation. This misses the whole point of the problem we tackle in this work, which relies on finding bugs in the protocol's implementation.

The tools presented in Section 3.3.2 [13–15, 18] have more similarities with the solution we propose to solve the problem posed in Section 1.1.

Zermia [18] implements a subset of fault with enough fault coverage to reproduce and find most bugs in BFT protocols. Moreover, other faults can be created by implementing the methods of an interface. However, it has two big downsides: *i*) it can only test protocols running in the JVM; *ii*) does not generate automatically the test cases.

Drăgoi et al. [14] and Winter et al. [15] present a fault injection tool that can automatically find bugs in BFT protocols. However, the faults implemented by these tools do not cover some of the behaviours leading to bugs in BFT protocols. Nevertheless, these works present useful techniques for generating test cases and consequently reduce the complexity of automatically finding bugs in consensus protocols.

Twins [13] automatically generates test cases containing the moments in which faults are injected, runs the test cases and verifies the consensus properties. Furthermore, it also implements a subset of faults with enough fault coverage to reproduce and find most of the bugs in BFT protocols.

Some criteria can be defined to assess and compare the quality of different fault injection tools. These tools can be used to evaluate: *i*) the performance of a system; or *ii*) its correctness. Some can evaluate both the performance and correctness of a system.

When evaluating the correctness of a system, the main objective is to detect and report bugs, together with enough information so that the user can understand its root cause and fix the respective vulnerability. Therefore, these should be reported with enough information so that they can be reproduced deterministically.

The subset of faults implemented by the fault injection tool must be as small as possible while having enough coverage to reproduce and find most of the bugs existing in BFT protocols. This reduces the complexity of automatically generating test cases, which helps significantly in finding new bugs without having to think about the sequence of steps of a protocol's execution that could lead to these bugs.

Systems are written in different programming languages and faults can be injected using different methods, such as code instrumentation. Depending on the instrumentation method, some fault injection tools can only test systems in certain runtimes. Others might require modifications or extensions to the source code to inject faults in a given system. The ideal case is to have a fault injection tool that is both language and protocol-agnostic.

Some fault injection tools test a modified version of the protocol's implementation, with some source code modifications. Others even test a different implementation, corresponding to a simulation of the original protocol. However, one can not assume that bugs found in these modified implementations translated directly to the real protocol implementation. Therefore, it is crucial for a fault injection tool to test the real protocol implementation.

Although we mainly aim to test the correctness of a protocol, it is equally important to assess its performance in the face of adverse network conditions. There can be cases in which the performance of the system is degraded so much that the system becomes unusable. This could lead to possible violations of the termination properties of a protocol, which we can only ensure given partial synchrony guarantees for most protocols. To properly ensure the correctness of these protocols' termination properties, it is fundamental to test their performance under different network conditions.

Given these insights to assess the quality of different fault injection tools, a more concise description of each of the criteria defined follows below. We use each of these criteria to compare the different fault injection tools discussed throughout this chapter in Table 1:

- **Reproducibility (REPR):** ***None*** in case the fault injection tool does not produce any output other than if there are bugs or not, or the produced output is not meaningful enough to reproduce faults; ***Low*** in case the output produced enables to reproduce faults stochastically; ***Med*** in case the output produced enables to reproduce faults non-deterministically; ***High*** in case the output produced enables to reproduce faults deterministically.

- **Automatic Fault Generation (AGEN):** ***Yes*** in case the fault injection tool generates the test cases containing the faults to be injected; ***No*** in case it is up to the user to define the moments in which these faults are injected.
- **Fault Coverage (FCOV):** ***Low*** in case it only includes faults such as Process Crashes, Message Delays, or Message Drops; ***Med*** in case it extends these faults with Message Modifications; ***High*** in case it extends these faults with some protocol-specific faults, such as Leader Equivocation, Forced Voting and Double Voting; ***Cust*** in case that in top of the previously mentioned faults it is possible to produce arbitrary semantically invalid messages.
- **Applicability (APPL):** ***SSpec*** (System Specific) in case the fault injection tool is built or can be extended specifically for a given system; ***RSpec*** (Runtime specific) in case the fault injection tool can only test systems in specific runtimes; ***Both*** in case it is specific for a given runtime and specific for a given system; ***Any*** in case the fault injection tool can test any system written in any programming language, without further extending its original code.
- **Real Implementation (REAL):** ***Yes*** in case the fault injection tool tests the real implementation of the system (its non-modified source code); ***No*** in case the fault injection tool tests a modified version of the system's source code (or a simulation of it).
- **Property Tested (PROP):** ***Perf*** (Performance) in case the fault injection tool is testing the Performance of the system; ***Corr*** (Correctness) in case the fault injection tool is testing the correctness of the system, such as the properties that should hold despite the faults that might occur. ***Both*** in case it tests both the Performance and Correctness of a system.

SYSTEM	REPR	AGEN	FCOV	APPL	REAL	PROP
FCatch [28]	High	Yes	Low	RSpec	Yes	Corr
Sieve [32]	High	Yes	Low	SSpec	Yes	Corr
Phoebe [33]	Low	Yes	Low	Both	Yes	Both
ChaosETH [34]	Low	Yes	Low	SSpec	Yes	Both
CEPPB [37]	Low	Yes	Med	Any	Yes	Perf
BFT Simulator [17]	High	No	Cust	SSpec	No	Both
BFT Bench [16]	High	No	High	SSpec	No	Perf
Zermia [18]	High	No	Cust	RSpec	Yes	Perf
TCICC [14]	Med	Yes	Low	SSpec	Yes	Corr
Random Test BFT [15]	Med	Yes	Med	SSpec	Yes	Corr
Twins [13]	High	Yes	High	SSpec	Yes	Corr
ByzPlug	High	No	Cust	Any	Yes	Corr

Table 3.1: Comparison of different fault injection tools and their capabilities. The capabilities of fault injection tools being compared are REPR: Reproducibility; AGEN: Automatic Fault Generation; FCOV: Fault Coverage; APPL: Applicability; REAL: Real Implementation; PROP: Property Tested.

Using the criteria defined above, we compare the quality of the different fault injection tools discussed

throughout this chapter in Table 1. Each criterion has a different importance in deciding whether a tool is more suitable to solve the problem posed in Section 1.1 than another.

First it is crucial that the real implementation of the system is being tested so that it is possible to reliably conclude something regarding that implementation's robustness. Faults injected should have enough coverage to reproduce most byzantine behaviours. The fault injection tool should not be limited to testing systems in a certain runtime, as this severely limits the different protocols the tool can be used with. It should not require re-implementing the necessary software to manipulate communication in a different system. Lastly, despite being crucial to finding bugs faster, automating the generation of test cases and verification of consensus properties is not a must. It is more important to have a deterministic way of reproducing bugs. Given these insights, there is only one fault injection tool that stands out from the others: Twins [13].

Twins stands out among the others, as the fault injection tool that gathers the best set of properties to reliably test the robustness of BFT protocols implementations. Therefore, we compare it with ByzPlug, giving relevance to the properties in which these differ, and assess how our work is novel and what improvements it introduces in this field of research.

Fault Coverage: ByzPlug implements *Message Drops*, *Message Modifications* and *Message Replays*, injected in UDP/IP and TCP/IP network packets independently of the protocol being tested, as detailed in Section 4.3. Section 4.6 explains how each fault injected in Twins can be implemented on top of these. Moreover, Twins authors argue that it is not interesting to test the protocol against semantically invalid messages and we disagree with this. In fact, a byzantine process can generate specially crafted messages that aim at exploring corner cases in the system behaviour, and a fault injection tool should provide the ability to study this behaviour. ByzPlug provides the ability to craft arbitrary semantically invalid messages, as explained in Section 4.6.

Automatic Testing: Twins ability to automatically test a protocol by: *i*) generating test cases; *ii*) running test cases; and *iii*) verifying the consensus properties is limited to protocols that share the same semantics as DiemBFT [6]. The same could be achieved using ByzPlug, by building the modules that achieve the same purposes as *(i)*, *(ii)* and *(iii)* specifically for protocols that share similar semantics. Using Twins to test a protocol with different semantics would require re-implementing all of those modules.

Applicability: While Twins requires extending a multiplexing wrapper (DiemBFT's Network Playground [43]) to intercept network traffic, ByzPlug can intercept and manipulate all process-level communication over UDP/IP or TCP/IP from any protocol without requiring any code extensions.

4

ByzPlug

Contents

In this chapter, we present our approach to build a tool to reliably assess the robustness of byzantine fault-tolerant protocols implementations. The tool should be able to test any protocol, independently of the programming language and libraries used to build it, without modifying or requiring extensions to its source code.

In Section 4.1 we start by giving an overview of the different programs ByzPlug is composed of. In each of the following sections we dive deeper into the details of each of these programs. In Section 4.2 we present the pseudocode of the `XDP` programs used to intercept and redirect network traffic. In Section 4.3 we describe the user space programs associated with each of the `AF_XDP` sockets, which make it possible to manipulate packets in user space. In Section 4.4 we describe the path of a packet from

the moment it is sent to the moment it is delivered, highlighting the different interception points and the actions taken at each of these. In Section 4.5 we explain how we inject faults despite the existence of Transport Control Protocol (TCP). Finally, in Section 4.6 we explain how we can access, modify and inject faults based on protocol-specific information of the intercepted packets.

4.1 Overview

ByzPlug is composed of a set of programs that are deployed to kernel and user space, for every process in a distributed system to be tested. The deployed programs work together to intercept and manipulate all process-level communication over UDP/IP and TCP/IP. Figure 4.1 gives an overview of the different programs that compose ByzPlug: *i*) eXpress Data Path [1] (XDP) programs that intercept arriving traffic; *ii*) C++ user-space programs associated with AF_XDP sockets [23] (XSK) to manipulate and inject faults in network packets; *iii*) an optional gRPC [22] server to manipulate protocol-specific data and inject faults based on that data.

For a distributed system to be tested using ByzPlug, each of its processes should be deployed inside a separate container and connected through a bridge. This can quickly be achieved by using a tool that facilitates the deployment of virtualized infrastructure, such as Docker [26].

Each container has a virtual ethernet interface with two ends (eth0 and veth), as explained in Section 2.3. To start testing a distributed system, for each container deployed, ByzPlug hooks an XDP program to each of its virtual ethernet interface ends and locally deploys an XSK for each of the XDP programs. Optionally a local gRPC server is also deployed for each of the system processes.

One XDP program per process would be enough to intercept all arriving network traffic, which could be manipulated and passed down to the kernel network stack. However, that would be done in a kernel space program, in which we would have to deal with the restrictions imposed by the eBPF verifier to manipulate the packet's data. Instead, we need to do this in user space, from where there is no support to re-inject the packet to the network stack. Our solution for this relies on deploying an XDP program at both ends of each virtual ethernet interface of each container, and redirecting the packet among both ends until it reaches the one it was originally intercepted in, where it is intercepted by the XDP kernel space program, and passed down to the network stack (XDP_PASS) after being manipulated on the eth0 in the middle of this traversal.

4.2 Kernel Space XDP programs

A packet arriving at one of the ends of the virtual interface is always intercepted by an XDP program hooked there. These programs are responsible for processing the packet headers and deciding what

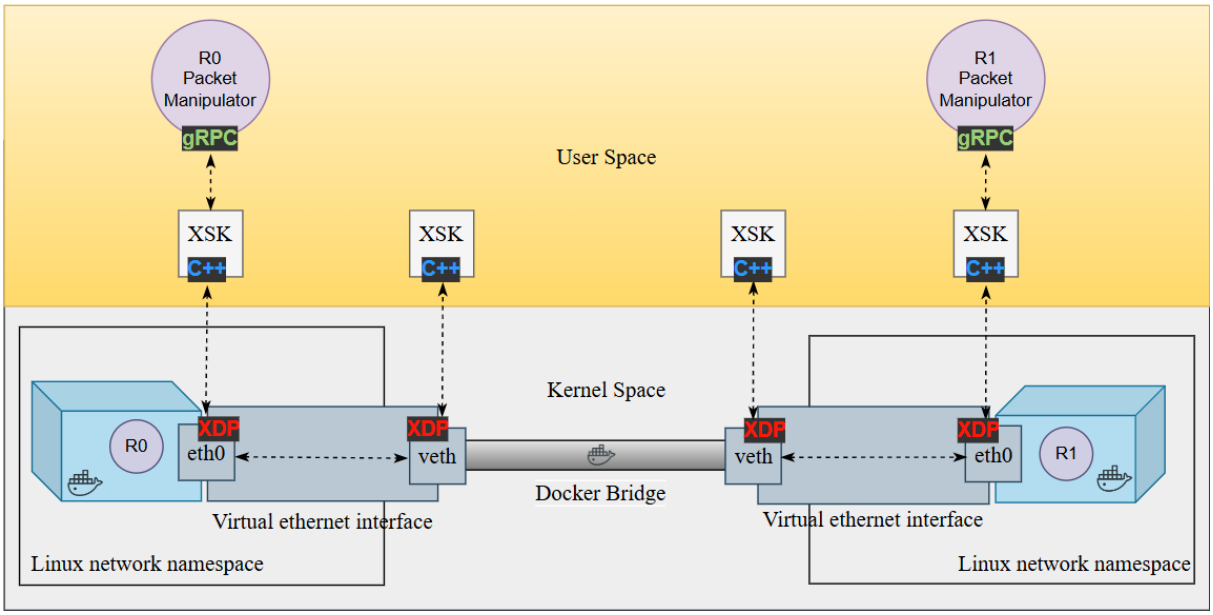

Figure 4.1: Overview of our fault injection tool. One XDP program attached in each virtual device of each node. One `AF_XDP` program associated with each of the XDP programs. An optional gRPC server handling the packet manipulator service call for each replica.

action should be taken, by returning one of the opcodes: *i*) `XDP_PASS` to pass the packet to the network stack; *ii*) `XDP_DROP` to drop the packet; *iii*) `XDP_REDIRECT` to redirect the packet to another network device.

For `XDP` programs to be able to distinguish the manipulated packet from the original packet the eth_protocol field is assigned some custom codes: `0x8000` is the default code of an IP packet; and `0x7999` represents an IP packet coming from the network. Both codes represent that the packet is still at the first interception point; `0x8421` indicates that the packet is in the second interception point, and not in the interface it was originally intercepted in; `0x8420` indicates that a packet is back to the interface it was originally intercepted in, ready to be passed to the network stack.

Algorithm 1 describes the action taken on each packet according to its eth_protocol value. The `XDP` programs re-assign the default IP packet code (`0x8000`) to the eth_protocol field before it is passed to the network stack, and the code (`0x7999`) after a packet is received from the network (*from_network()*). Meanwhile, the user space programs described in the Section 4.3 are responsible for swapping the codes `0x8000` and `0x7999` for `0x8421`, and `0x8421` for `0x8420` signalling the end of packet manipulation. Figure 4.2 illustrates how the eth_protocol code changes along the packet's path from a sender (R0) to a receiver (R1).

Algorithm 1 also shows that some packets coming from the network (*from_network() - line 2*), classified as acknowledgements(*is_ack()*) should be dropped (*should_drop()* - line 4). This is the case for acknowledgements of replayed packets, as mentioned in Section 4.5.2. Finally, acknowledgements leaving to the network should just be passed down to the network stack (*line 12*).

Algorithm 1: Pseudocode of XDP Programs hooked to devices on Virtual Ethernet Interfaces

```
1  if ethhdr_protocol == 0x8000 then
2      if from_network(ethhdr) then
3          if is_ack(tcphdr) then
4              if should_drop(ack_num) then
5                  return XDP_DROP;
6              else
7                  return XDP_PASS;
8          ethhdr_protocol = 0x7999;
9          return XDP_REDIRECT(xsks_map);
10     else
11         if is_ack(tcphdr) then
12             return XDP_PASS;
13         else
14             return XDP_REDIRECT(xsks_map);
15 else if ethhdr_protocol == 0x8421 then
16     return XDP_REDIRECT(xsks_map);
17 else if ethhdr_protocol == 0x8420 then
18     ethhdr_protocol = 0x8000;
19     return XDP_PASS;
20 return XDP_PASS;
```

4.3 User Space AF_XDP programs

As mentioned in Challenge 5 we need to redirect packets to user space, where we can manipulate them without dealing with the restrictions imposed by the eBPF verifier. To do so, we associate with each of the XDP programs an AF_XDP socket, and with each of the sockets a user space program.

The user space programs associated with each XSK all perform the same set of actions, presented in detail in Section 2.8. These continuously wait for packet frames redirected from its associated XDP program. When a frame is accessible, its eth_protocol is changed accordingly and possibly a fault is injected. ByzPlug implements three different fault primitives, which are agnostic to the system being tested:

- **Message Omission:** the frame is removed from the RX ring and not placed on the TX ring.
- **Message Replay:** the frame descriptor is only removed from the TX ring after the *sendmsg()* system call is invoked as many times as the packet should be replayed.
- **Message Modification:** the frame is removed from the RX ring and its application data is modified. The size of the frame is updated before being placed on the TX ring.

This implementation of faults works fine when the transport protocol is UDP, in which the only detail to be taken into account is to set the checksum to zero before placing the packet in the TX ring. However,

when TCP is used some additional steps must be performed, which we explain in depth in Section 4.5. To avoid concurrency problems, ByzPlug only allows injecting faults in packets before these reach the network, at the `eth0` end.

4.4 Packet Flow

Figure 4.2 illustrates the packet flow, with an example of a packet sent from R0 to R1. Furthermore, it shows how the eth_protocol values change along its path: *1*) When a packet is intercepted for the first time (in the `veth` device) the value of its eth_protocol field is the default code for an IP packet of `0x8000`; *2*) In the user space program the packet gets assigned the code `0x8421`, and gets sent to the other end (`eth0`); *3*) The XDP program intercepts the packet, and according to Algorithm 1 redirects it to userspace, where it is assigned the code `0x8420` and sent back to `veth` (the device it was initially intercepted in) after being manipulated; When intercepted again at `veth` device the packet's code is changed back to `0x8000`, as specified in Algorithm 1. *4*) as the packet arrives from the bridge and is intercepted in the `eth0` device of the receiver, it has the code `0x8000`. However, because the destination fields match the receiver host information (classified by *from_network()* function used in Algorithm 1), it is classified as coming from the network and the code `0x7999` is assigned to the packet before it is sent to user space; *5*) at this stage, its application data can be observed in user space before it is redirected to the other end (`veth`) of the virtual ethernet interface with the code `0x8421`; *6*) as the packet is intercepted with the code `0x8421` by the `XDP` program, it is sent to the other end (`eth0`) with the code `0x8420` through the user space program; *7*) finally, when intercepted again at the end it was initially intercepted with the code `0x8420`, the packet's eth_protocol value is changed to `0x8000` and the packet is passed to the network stack.

Two optional steps can be added to the path of the packet, in which the eth_protocol code does not change. The purpose of those steps is to observe protocol-specific data, or to manipulate and inject faults in the packet according to that data. Both steps can be performed as the packet is intercepted in the `eth0` virtual device of each container. In Section 4.6 we give an in-depth description of what these optional steps consist of.

4.5 Generic Fault-Injection

ByzPlug library implements three faults, injected at the network packet level: *Message Omission*, *Message Modification* and *Message Replay*. Built upon these, other faults can be injected, such as *Leader Equivocation*, *Forced Voting* or *Semantically Invalid Messages.* However, these require protocol-specific information. We explore how we can access and manipulate protocol-specific information in Section 4.6.

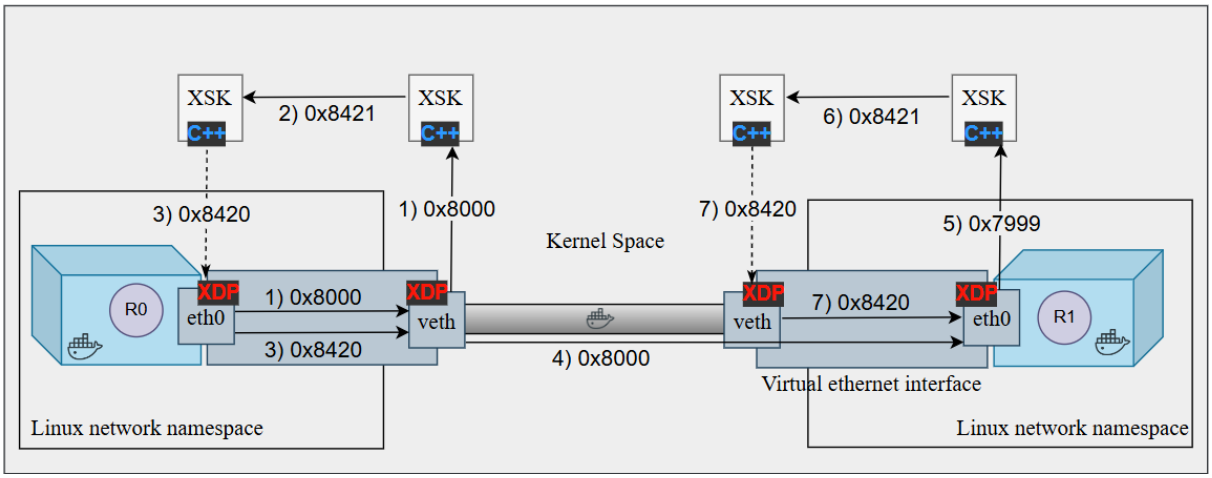

Figure 4.2: Overview of packet flow and its eth_protocol value changes. Packet is sent from process R0 to process R1.

A *Process Crash* fault can be performed by dropping all the messages sent and delivered to a given process.

We argue that these three implemented byzantine behaviours are enough to cover a wide variety of internal bugs in the protocol's node code. Despite many errors that can manifest internally and externally in a process, it is the interaction between the different protocol nodes that can lead to the invalidation of a consensus property. Correct nodes can either stop deciding or decide something different from other correct nodes, by being misled by a byzantine node. Therefore, the potential behaviours can be abstracted in two ways: whether a node is still participating in the protocol, and how it is participating in the protocol (what is the content of its messages and how it handles replayed messages). As a result of this abstraction, we implement these three faults.

The implementation of these faults should take into account that Transmission Control Protocol (TCP) might be used to establish reliable bidirectional communication between the protocol nodes.

UDP does not have any mechanism to ensure reliable communication. Therefore, when it is used as the transport protocol, messages can be freely dropped and replayed. The only concern is to set the checksum to zero when Message Modification faults are injected.

TCP provides a more interesting discussion and requires a more careful approach to be able to inject each of these three faults. It ensures that: *i*) No message with sequence number N is processed by the application until all the messages with sequence number $N' < N$ have been processed; *ii*) Messages that have not been acknowledged are considered lost and consequently re-transmitted until acknowledged; *iii*) A sender only considers its message with sequence number N and size S acknowledged after it receives a packet with the ACK flag and the acknowledgement number $N+S$.

In the following subsections, we describe how we convince TCP that the communication is reliable,

despite the injection of these different faults.

4.5.1 Message Omission

Message Omission consists of dropping a packet that should be delivered to a receiver.

Consider a scenario in which a packet with *len* application data bytes and sequence number *S* is sent from a node R0 to a node R1. R0's TCP state machine increments its next sequence number by *len* to *S' = S + len* and expects an acknowledgement packet with the acknowledgement number of *S'*. Given that the packet is dropped before reaching the network, R1 does not acknowledge this message.

This has two consequences: *i*) the sender (R0) will re-transmit this message until it receives a packet from that receiver (R1) with an acknowledgement number $S'' \geq S'$; *ii*) the receiver (R1) will not process the subsequent packets with sequence number $S'' \geq S'$ because it has not yet received the packet with sequence number *S* from R0.

Consider Δ as the total number of application data bytes dropped in messages sent from R0 to R1. To solve consequence *(i)* every subsequent packet acknowledged by this receiver has its acknowledgement number incremented by Δ. Similarly, to solve consequence *(ii)* every subsequent packet sent from R0 to R1 has its sequence number decremented by Δ.

Figure 4.3 describes a scenario in which one Message Omission fault is injected. A TCP packet with 10 bytes of application data and sequence number 0 is sent from R0 to R1. On its way to R1, it is dropped at R0's eth0 device and Δ is incremented by the number of application data bytes (10) of the dropped packet. Another packet is sent from R0 to R1, now with a sequence number of 10 and 10 application data bytes, to which no fault is injected. Before it leaves from R0's eth0 device to the network the packet's sequence number is decremented by Δ. R1 receives a packet with sequence number 0 and 10 application data bytes. R1 acknowledges the message received, by sending a packet with acknowledgement number 10. The acknowledgement packet is intercepted at R0's eth0 device by an `XDP` program. Before being processed by the network stack the acknowledgement number of the packet is incremented by Δ from 10 to 20. This gives R0 the illusion that R1 has processed all the packets that R0 has sent until now, including the dropped packet, when in reality R1 has not processed the dropped packet, as supposed.

4.5.2 Message Replay

Message Replay consists of replaying the transmission of a certain packet to a receiver more than once.

Consider a scenario in which a packet with *len* application data bytes and sequence number *S* is sent from a node R0 to a node R1. R0's TCP state machine increments its next sequence number by *len* to *S' = S + len* and expects an acknowledgement packet with the acknowledgement number of *S'*.

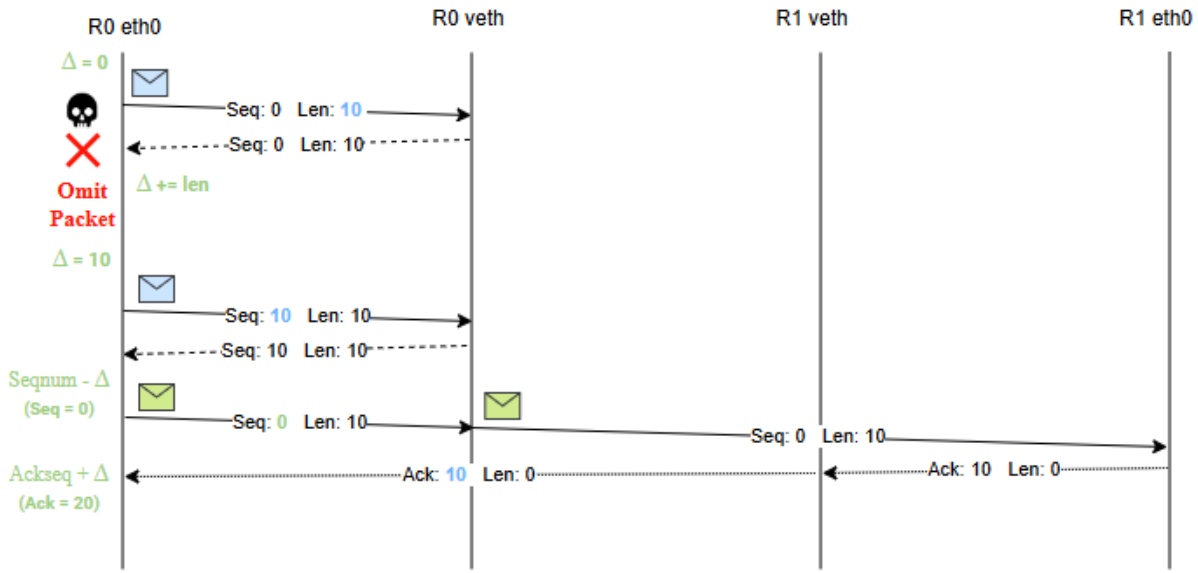

Figure 4.3: Message Omission fault injected in a TCP packet with sequence number 0 and 10 application data bytes.

In case the packet is replayed once, R1 receives two packets with the same sequence number *S*. The first packet with sequence number *S* is processed and an acknowledgement is sent back to R0 with sequence number *S'*. However, because R1's TCP state machine is now expecting a packet with sequence number *S'*, it flags the second packet as a TCP Retransmission and does not process it.

For R1 to process the replayed packet its sequence number should be incremented by *len* from *S* to *S'* before getting sent out to the network. However, this leads to three consequences: *i*) R0's TCP state machine knows that R0 only sent one packet with sequence number *S* and *len* application data bytes. As R1 acknowledges the replayed packet, it sends a packet with acknowledgement number $S'' = S' + len$, which is not expected by the sender (R0). Therefore R0 flags the 'TCP Acked unseen segment' error and re-transmits the last sent packet; *ii*) the next packet sent from R0 to R1 will carry a lower sequence number (*S'*) than expected by R1 (*S"*). As a result, R1 will re-acknowledge the last TCP segment seen. *iii*) the next acknowledgement of a non-replayed packet sent from R1 to R0 will carry an acknowledgement number *len* * *N* times higher than expected. *N* is the amount of times that the packet is replayed. R0 flags the 'TCP Acked unseen segment error' and re-transmits the last sent packet.

To solve consequence *(i)* every acknowledgement of a replayed packet sent from R1 to R0 should be dropped. To achieve this, before sending a replayed packet with sequence number *S* and *len* application data bytes, the acknowledgement number *S'* = *S* + *len* should be stored in an eBPF map. Because the map is shared between the user and kernel space, the `XDP` program that intercepts arriving packets can drop acknowledgement packets in case there is an entry in that eBPF map with that acknowledgement number as a key (classified by *is_ack()* and *should_drop()* in Algorithm 1).

Consider Δ as the total number of application data bytes replayed in messages sent from R0 to R1. To solve consequence *(ii)* every time a message with *len* application data bytes is replayed *N* times, Δ should be incremented with the value *N* * *len*. Subsequently, every packet sent from R0 to R1 should have its sequence number incremented by Δ. To solve consequence *(iii)* every subsequent acknowledgement packet sent from R1 to R0 should have its acknowledgement number decremented by Δ. An important detail is that this Δ should only be incremented after the packet is replayed the number of times specified in the fault to be injected.

Figure 4.4 describes a scenario in which one Message Replay fault is injected. The message is replayed only once. A packet with sequence number 0 and 10 application data bytes is sent from R0 to R1. This packet is replayed once at R0's eth device, and the number 20 is stored in the eBPF map that keeps track of the acknowledgement numbers to be dropped. Consequently, Δ is incremented by the number of application data bytes (10) replayed multiplied by the number of times the packet was replayed (1). R1 receives the original packet with sequence number 0 and sends back an acknowledgement packet with number 10, which is processed by R0. R1 receives the replayed packet with sequence number 10 and sends back an acknowledgement packet with number 20. The packet is intercepted and dropped by the `XDP` program hooked to R0 eth0's device, and an entry in the eBPF map is found with the acknowledgement number 20. Another packet is sent from R0 to R1 and no fault is injected on this packet. Before being sent out to the network, this packet's sequence number gets incremented by Δ from 10 to 20. R1 receives a packet with sequence number 20 and 10 application data bytes, sending back a packet with acknowledgement number 30. The `XDP` program intercepted at R0 eth0's device intercepts this packet and decrements its acknowledgement number by Δ before it is processed by its network stack.

4.5.3 Message Modification

Message Modification consists of modifying the bytes of a packet.

Consider a scenario in which a packet with *len* application data bytes and sequence number *S* is sent from a node R0 to a node R1. R0's TCP state machine increments its next sequence number by *len* to *S'* = *S* + *len* and expects an acknowledgement packet with the acknowledgement number of *S'*.

In this scenario, a Message Modification fault is injected and the packet bytes are modified. The new packet has *len'* application data bytes. R1 acknowledges this packet by sending a packet with acknowledgement number *S** = *S* + *len'*.

Consider δ as the difference between the number of bytes from the modified packet and the original packet (δ = *len'* - *len*). Depending on the value of δ, three possible scenarios unfold from this:

- δ = 0: if the modified bytes differ from the original bytes R1 receives a packet with the wrong checksum, consequently not processing it.

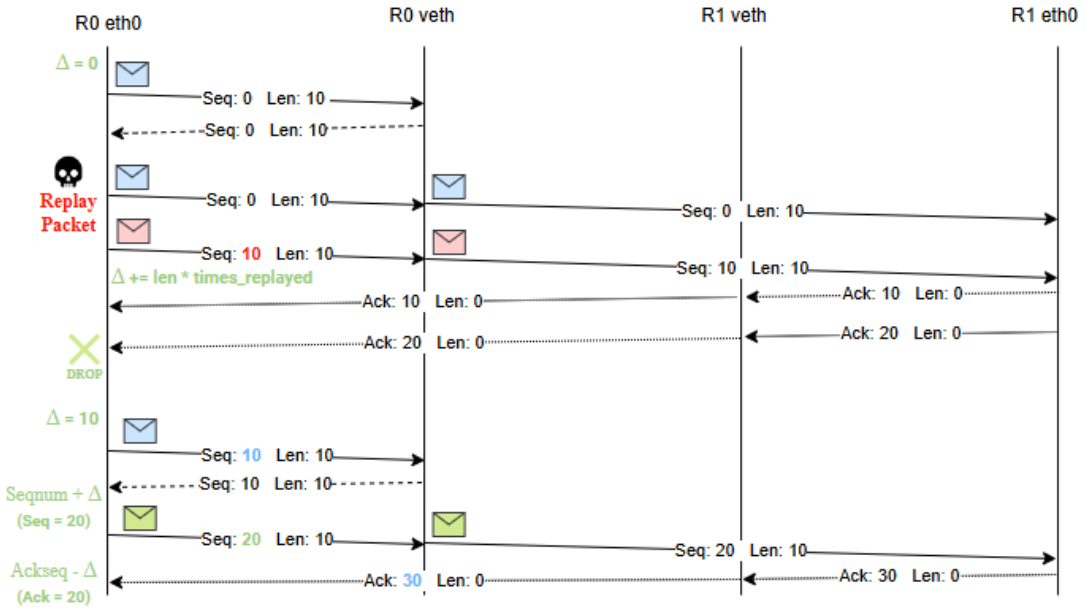

Figure 4.4: Message Replay fault injected in a TCP packet with sequence number 0 and 10 application data bytes. The message is replayed only once.

- $\delta > 0$: R1 receives a packet with δ more bytes than sent by R0. This leads to two consequences: *i*) R1 sends an acknowledgement with a number higher δ units than expected by R0 $S^* > S'$, leading to the 'TCP Acked unseen segment' flags; *ii*) the next sequence number sent by R0 is δ units lower than the expected by R1. R1 will consequently re-acknowledge the last seen segment, leading to 'Duplicate Acknowledgement' flags.

- $\delta < 0$: R1 receives a packet with δ less bytes than sent by R0. This leads to two consequences: *i*) R1 sends an acknowledgement with a number lower δ units than expected by R0 $S^* < S'$. This leads R0's TCP State machine to start re-transmitting the last segment sent until the expected acknowledgement number is received. *ii*) the next sequence number sent by R0 is δ units higher than the expected by R1. R1 will flag this as 'TCP out-of-order segment' and not process it.

All the scenarios require the update of the different protocol headers checksum after modifying the packet bytes. The second and third scenario require some additional effort. Nevertheless, with the use of this δ variable, the two consequences of both scenarios can be solved at the same time.

Consider Δ as the sum of all δ from messages sent from R0 to R1. To solve consequence *(i)* every subsequent acknowledgement packet sent from R1 to R0 gets decremented Δ before being processed by the network stack. Similarly, to solve consequence *(ii)* every subsequent packet sent from R0 to R1 gets its sequence number incremented by Δ before being sent out to the network.

Figure 4.5 and Figure 4.6 describe a scenario in which a Message Modification fault is injected. In

Figure 4.5 the modified packet has more bytes than the original, while in Figure 4.6 the modified packet has fewer bytes than the original. Respectively, these represent both cases in which δ is positive and negative.

In Figure 4.5 a packet is sent from R0 to R1 with sequence number 0 and 10 application data bytes. Before being sent out to the network a Message Modification fault is injected. Instead of the original packet, a modified packet with sequence number 0 and 20 application data bytes is sent out to the network (δ = 10). R1 receives the modified packet and sends back an acknowledgement packet with the number 20. This acknowledgement is intercepted by the `XDP` program hooked to R0's eth0 device, which decrements this number by Δ (10) before the network stack processes this packet with the acknowledgement number 10. R0 sends another packet to R1 and no fault is injected in this packet. This packet carries the sequence number 10 and 10 application data bytes. Before it is sent out to the network, the packet's sequence number is incremented by Δ (10). This packet is sent out to the network with sequence number 20 and 10 application data bytes, eventually reaching R1. R1 sends an acknowledgement to R0 with the number 30. Again, before being processed by the network stack the packet has its acknowledgement number decremented by Δ (10) to the number 20.

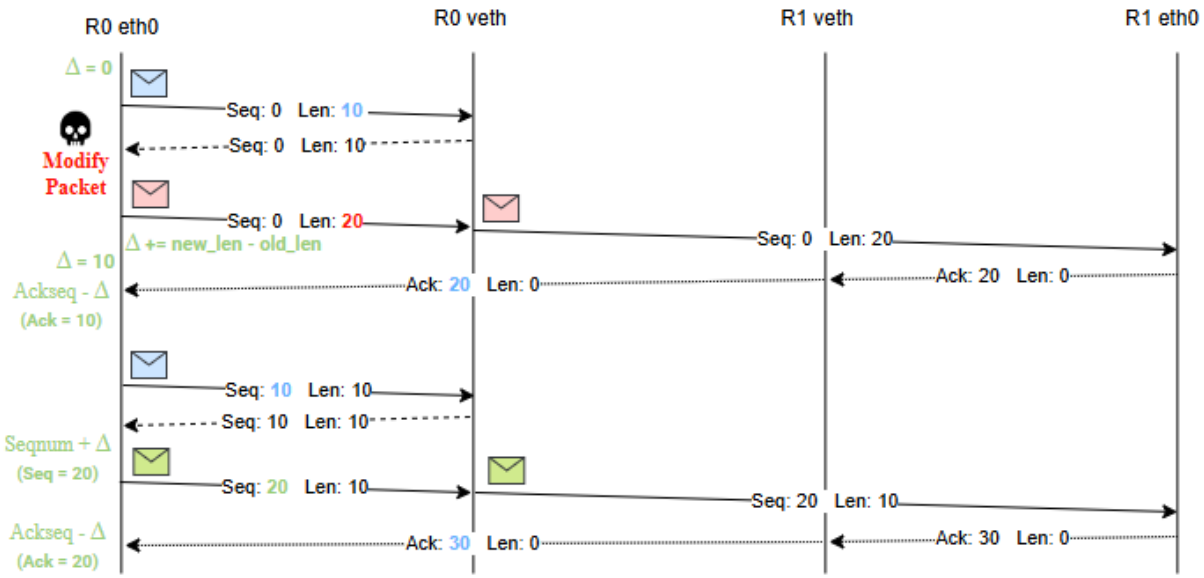

Figure 4.5: Message Modification fault injected in a TCP packet. The modified message has more bytes than the original message ($\delta > 0$).

In Figure 4.6 a packet is sent from R0 to R1 with sequence number 0 and 10 application data bytes. Before being sent out to the network a Message Modification fault is injected. Instead of the original packet, a modified packet with sequence number 0 and 5 application data bytes is sent out to the network (δ = -5). R1 receives the modified packet and sends back an acknowledgement packet with the number 5. This acknowledgement is intercepted by the `XDP` program hooked to R0's eth0

device, which decrements this number by Δ (-5) before the network stack processes this packet with the acknowledgement number 10. R0 sends another packet to R1 and no fault is injected in this packet. This packet carries the sequence number 10 and 10 application data bytes. Before it is sent out to the network, the packet's sequence number is incremented by Δ (-5). This packet is sent out to the network with sequence numbers 5 and 10 application data bytes, eventually reaching R1. R1 sends an acknowledgement to R0 with the number 15. Again, before being processed by the network stack the packet has its acknowledgement number decremented by Δ (-5) to the number 20.

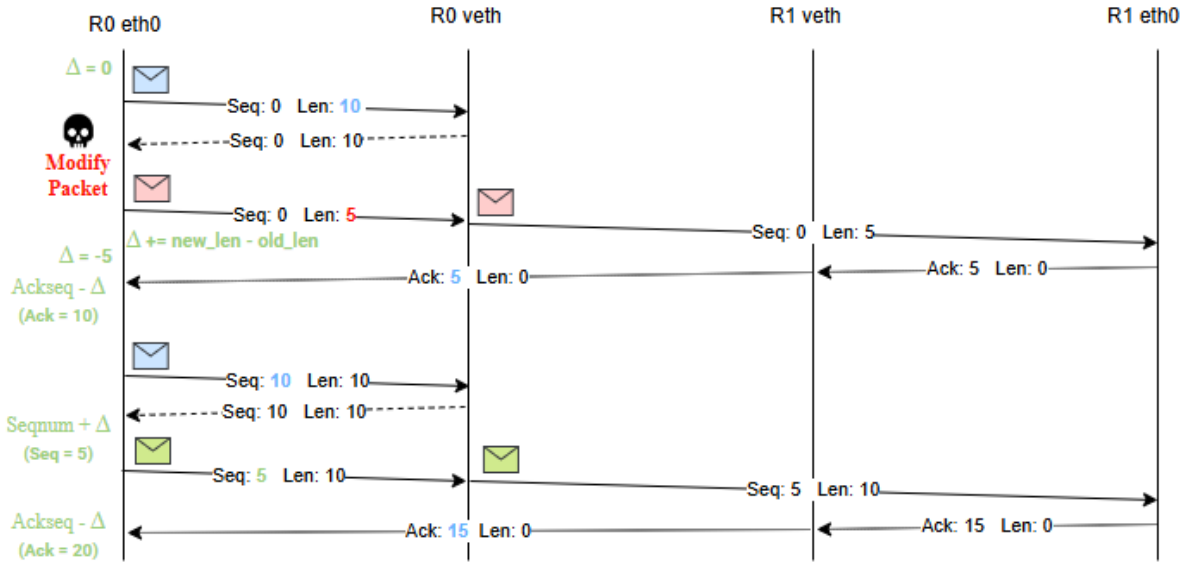

Figure 4.6: Message Modification fault injected in TCP packet. The modified message has less bytes than the original message ($\delta < 0$).

4.6 Protocol-Specific Fault-Injection

In the previous section, we described the different faults implemented by our library, and how these can be injected despite the existence of a transport protocol such as TCP. However, these are just primitives that work with plain packet bytes. To extract and manipulate protocol-specific information we need to convert these bytes into some data structure defined in the protocol, a process called deserialization. This allows us to *i*) inject faults based on protocol-specific information, instead of random moments of its execution; *ii*) implement other faults on top of the Message Modification primitive, such as Leader Equivocation, Forced Voting or other semantically invalid messages.

Leader Equivocation consists of having a node that is not the leader sending proposals as if it were the leader. This fault can be injected by changing any message sent from a node that is not the leader to a proposal. This requires knowing the structure of a proposal, which is specific to a given protocol.

Forced Voting consists of having a node vote on a proposal that it would not vote according to its internal state. This fault can be injected by changing any message sent from a node to a vote. Similarly, this requires knowing the structure of votes and distinguishing the different messages, which are specific to each protocol.

In both of these cases we want to preserve the original messages sent by the protocol nodes, while either sending an additional `Proposal` (as if the node is the leader) or forcing a node to send a `Vote` for a `Proposal` that it would not vote according to its state. Some protocols exchange `Ping-Pong` messages to keep a connection alive. In these cases, a `Ping` message should be changed for a proposal or a vote, instead of any other protocol message, preserving the original messages sent by the protocol nodes.

Double Voting consists of injecting a *Message Replay* primitive in a message that is a protocol vote.

Users can get creative when crafting *Semantically Invalid Messages*, either by corrupting signatures, proposed values, the identity of the sender, the structure of messages, view numbers or other specific fields.

ByzPlug allows accessing, modifying and injecting faults based on protocol-specific information with any programming language that is compatible with Protocol Buffers [21], which we use to define a packet manipulator service shown in Algorithm 2. To do so, after receiving an intercepted packet, a user space program `XSK` program has to make an `RPC` call to a `gRPC` server implementing the packet manipulator service. This server should be deployed on the host namespace. The packet bytes are sent to the `gRPC` server, together with some additional information from and deducted from the packet's frame (Algorithm 2).

In the `gRPC` server the packet bytes can be deserialized to a protocol-specific message. The user can modify the message arbitrarily (including creating a new message with a different structure from scratch) and decide whether to inject faults based on any of its fields, or any other information gathered from previous messages. Indications on which faults to inject are returned in the response to the RPC call, together with the serialized bytes of the possibly modified content of the packet to be delivered to the destination.

The `RxContent` message is the input to the service call, containing the information from the packet frame retrieved from the `RX` ring in the user space program: *i*) the application data of the packet (in bytes); *ii*) an unsigned integer representing the size of the application data in bytes; *iii*) indication of whether the packet is coming from the network (*incoming*=true); *iv*) protocol replica id of the sender of the packet; *v*) protocol replica id of the destination of the packet.

The service call returns the `TxContent` message, containing the information of the packet frame to be sent out to the network through the `TX` ring after manipulation: *i*) the possibly modified application data in bytes; and indications of whether any fault should be injected based on this data (*ii*) whether the packet bytes should be replaced; *iii*) the number of times the packet should be replayed; and *iv*) whether

the packet should be dropped).

Algorithm 2: Protocol Buffers definition of Packet Manipulator Service and its messages.

```
1  message RxContent {
2      required bytes content = 1;
3      required uint32 size = 2;
4      required bool incoming = 3;
5      required uint32 srcrid = 4;
6      required uint32 destrid = 5;
7  }
8  message TxContent {
9      required bytes content = 1;
10     required bool modified = 2;
11     required uint32 replay = 3;
12     required bool omit = 4;
13 }
14 service PacketManipulator {
15     rpc Manipulate (RxContent) returns (TxContent);
16 }
```

5

Evaluation

Contents

In this chapter we evaluate ByzPlug by assessing *i*) whether it is capable of reproducing bugs; and *ii*) what is the performance overhead of having full observability over all process-level communication of a distributed system.

Our target application consists of a C++ implementation [20] of the Two-Chain variant of the HotStuff protocol. In Section 5.1 we expose bugs found using ByzPlug and give instructions on how to reproduce them. In Section 5.2, we evaluate the performance overhead of having full observability over all process-level communication of this HotStuff implementation. The performance metrics are CPU Usage, memory usage, throughput and latency. These metrics are compared between a setup in which ByzPlug is intercepting and deserializing protocol messages, and a vanilla setup, in which nothing else besides the protocol is running.

We conducted our experiments (both performance evaluation and bug reproduction) in a machine with 128GB RAM and two Intel(R) Xeon(R) Gold 5320 processors with 52 cores. All cores sustain a clock speed up to 2.2GHz. In this machine, we set up a vagrant virtual machine with 32 cores and 32GB

RAM with Ubuntu 22.04 LTS distribution. All replicas are run on the same virtual machine in the same virtual network. The virtual network and replicas are deployed using Docker.

5.1 Bug Reproduction

During this research, we found a bug in an implementation [20] of the Two-Chain variant of HotStuff [7]. The bug found consists of a safety violation, in which a correct node unintentionally misleads other correct nodes to commit a modified block, previously proposed by a byzantine node. The committed block contains a *command* value arbitrarily chosen by the byzantine node, which does not correspond to any *command* sent by a client.

5.1.1 Safety Violation on Two-Chain Hotstuff

To reproduce this safety violation, we instantiate four HotStuff replicas (R0, R1, R2 and R3), and set R0 as byzantine. To reproduce the bug, we deterministically control the leader election, the number of proposals of each view and the trigger to change the view, by implementing a separate HotStuff module referred to as Pacemaker. However, the same bug would manifest with the existing *RoundRobin* Pacemaker implemented in HotStuff source code, or other Pacemaker implementations that allow the following leader schedule: [R0, R0, Rn, Rn], where Rn is any node to which the byzantine node proposes the modified block.

Briefly, the steps to reproduce the bug are: *i*) byzantine node gathers a QC for a proposed block b1; *ii*) byzantine node poisons the state of a correct node with a modified block, which carries the QC of block b1 but an arbitrary *command* value; *iii*) the node with the poisoned state gathers a QC for a block, which carries the QC of block b1 but extends the modified block; *iv*) the node with the poisoned state gathers a QC for another block it proposes, forming a Two-Chain.

After this sequence of steps, when the node with the poisoned state receives or sends a proposal, all the non-committed blocks up to (and excluding) its highest QC are committed. Included in these non-committed blocks there is the modified block with an arbitrary *command* value. A safety violation occurs, as every correct node commits this block.

Each of these steps can be represented by a view. Figures 5.1 to 5.4 illustrate the sequence of events on each of these views, from the messages sent and received to the faults injected. In these figures, the yellow blocks represent committed blocks and the blue blocks are the ones for which a QC has been obtained. The orange colour on replicas represents the byzantine replicas and the purple represents correct replicas. A Two-Chain is formed when two consecutive blocks are blue, just like in Figure 5.5. Below we give an in-depth description of the events on each view:

View 1: [1] The Pacemaker elects R0 as the leader of View 1. R0 proposes b1 to all replicas and gathers votes from all replicas. No faults are injected. R0 gathers the QC for b1 and a view change is triggered.

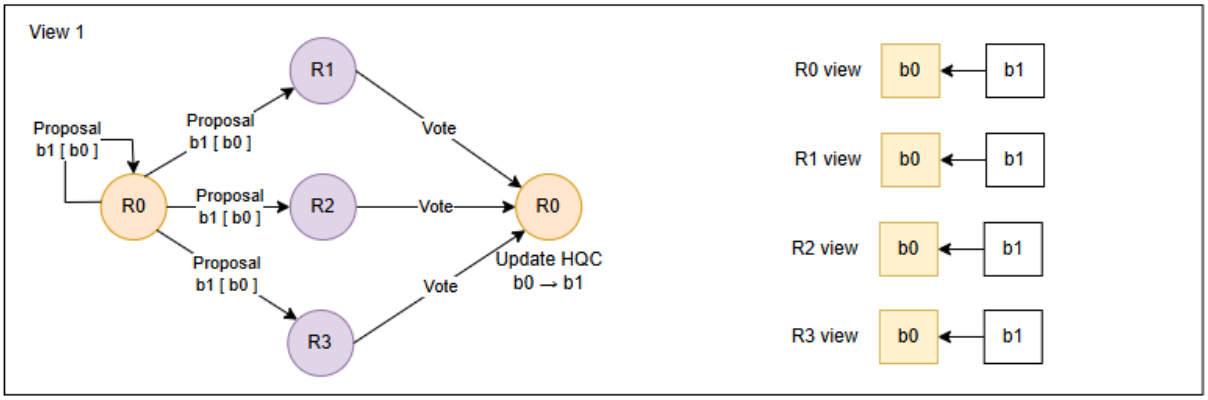

Figure 5.1: View 1 of Two-Chain HotStuff safety violation. R0 is the leader and the byzantine node.

View 2: The Pacemaker elects R0 as the leader of View 2. R0 proposes a block (b2) to R1 and a modified block (b2') with an arbitrary *command* value (b2') to R2. R0 gathers a vote from R1 and itself. The vote from R2 to R0 is dropped, to mimic R0 ignoring the vote on this modified block. We drop the proposal from R0 to R3 to stop progress on this view. [2]

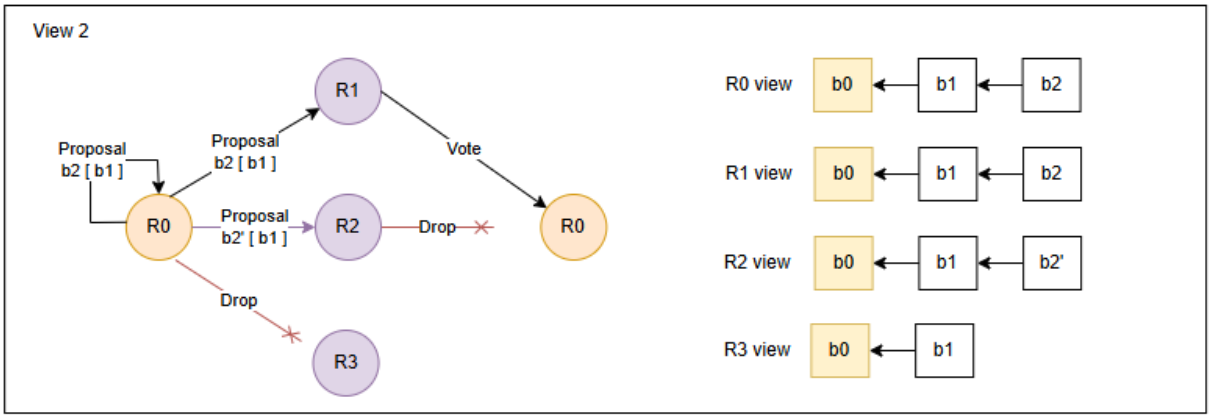

Figure 5.2: View 2 of Two-Chain HotStuff safety violation. R0 is the leader and the byzantine node.

View 3: The Pacemaker elects R2 as the leader of View 3. R2 proposes b3, extending the highest QC it knows about (b1) and extending a chain containing the modified block with an arbitrary *command* value (b2'). Every replica votes on this proposal, and R2 updates its highest QC to b3.

[1] This view is included to avoid starting the attack at the genesis block. However, it is not necessary to successfully produce this safety violation.

[2] It is not necessary to avoid progress on this view, as we control the leader election deterministically. However, we do it because HotStuff's RoundRobin Pacemaker would trigger a view change and rotate the leader in this case.

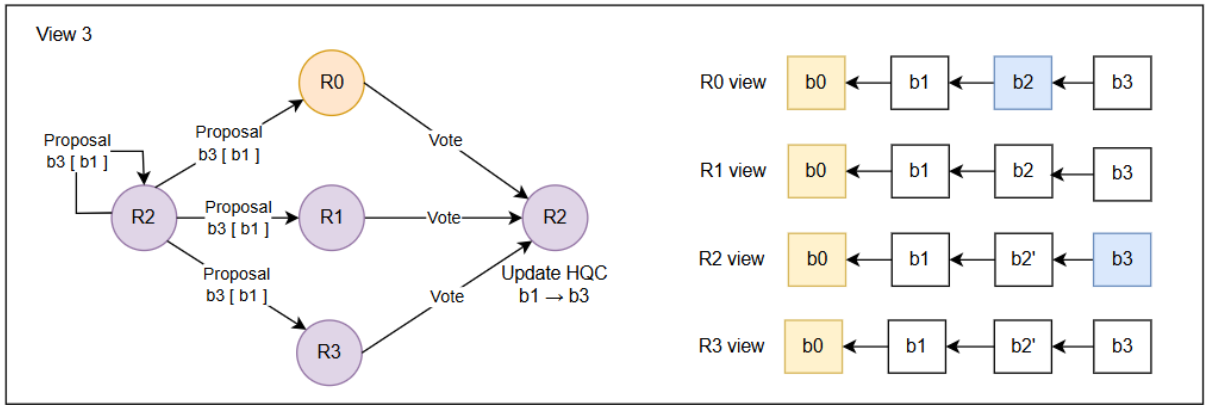

Figure 5.3: View 3 of Two-Chain HotStuff safety violation. R2 is the leader and R0 is the byzantine node.

View 4: The Pacemaker elects R2 as the leader of View 4. R2 proposes b4, extending the highest QC it knows about (b3). Every replica votes on this proposal, and R2 updates its highest QC to b4.

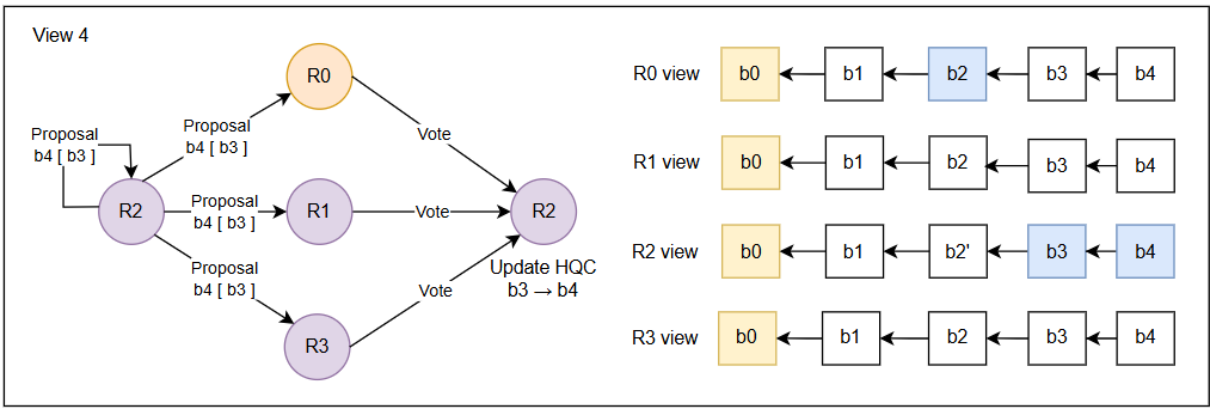

Figure 5.4: View 4 of Two-Chain HotStuff safety violation. R2 is the leader and R0 is the byzantine node..

View 5: At this point a Two-Chain is formed: b2' is a direct parent of b3, for which a QC is gathered pointing to b2'; and b3 is a direct parent of b4, for which a QC is gathered pointing to b3. Whenever R2 receives or sends a proposal, R2 and other correct replicas will commit b1, b2 and b3. A safety violation is produced as b2' is committed by correct replicas.

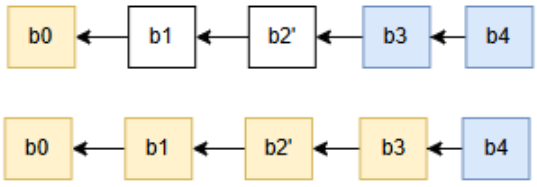

Figure 5.5: View 5 of Two-Chain HotStuff safety violation. Correct replicas commit b1, b2' and b3.

5.2 Evaluating Performance

The performance metrics assessed are CPU Usage, memory usage, throughput and latency.

CPU Usage is presented in two ways: *i*) differentiated between system CPU and user CPU; and *ii*) the sum of both. Memory usage is presented as the sum of the memory used by RAM, Cache and Kernel buffers. Throughput is defined as the number of finalized client requests per second. To compute this the protocol is run for 10 minutes and the *command* height of the last commited *command* is divided by this time in seconds. Latency is defined as the average time to finalize a client request, measured using the salticidae C++ library [44].

We run four replicas in a configuration that tolerates a single byzantine replica. We fix one of the replicas as the leader for the entire duration of the experiment, to avoid additional overhead. We run four clients that send *command* requests during 10 minutes to the leader. Each client has a maximum of eight pending asynchronous *command* requests at once.

The results of evaluating the CPU usage are displayed in Figures 5.6 and 5.7, and memory usage is plotted in Figure 5.8 [3]. We report the metrics in two different configurations: *i*) a vanilla setup, in which only the protocol nodes are executed; and *ii*) a setup in which every message is intercepted by ByzPlug and deserialized to a protocol-specific data structure.

The results from Figure 5.6 regarding CPU Usage show that setup ByzPlug has an overhead of around 1.76x over the vanilla configuration. A more detailed view of the CPU Usage can be seen in Figure 5.7, in which the user CPU and system CPU are differentiated. We can see how the system CPU is lower using ByzPlug to intercept all messages, than in a vanilla configuration. However, user CPU has an overhead of around 2.3x. The reason for this is because packet processing is not being done by the kernel network stack, but instead by the user space `XSK` programs described in Section 4.3.

The results from Figure 5.8 regarding memory usage show that ByzPlug uses 1.12x more memory than the vanilla configuration. It is important to highlight that the CPU overhead consists of the total overhead from intercepting network traffic of the four replicas, running in the same Virtual Machine, and would be 4x lower in the case of a real distributed system.

Regarding throughput and latency, with the vanilla configuration it is possible to commit 15.7 decisions per second, with an average latency of 1.97 seconds to commit each decision. With ByzPlug it is possible to commit 6.6 decisions per second with an average latency of 4.9 seconds to commit each decision. Consequently, ByzPlug degrades throughput and latency on an average of 2.4x, meaning that the protocol will commit decisions 2.4 times slower.

[3] There is a small bump in Memory and CPU usage in the vanilla configuration at the exact time, which should be due to a process that spuriously launched at that time and consumed some memory and CPU.

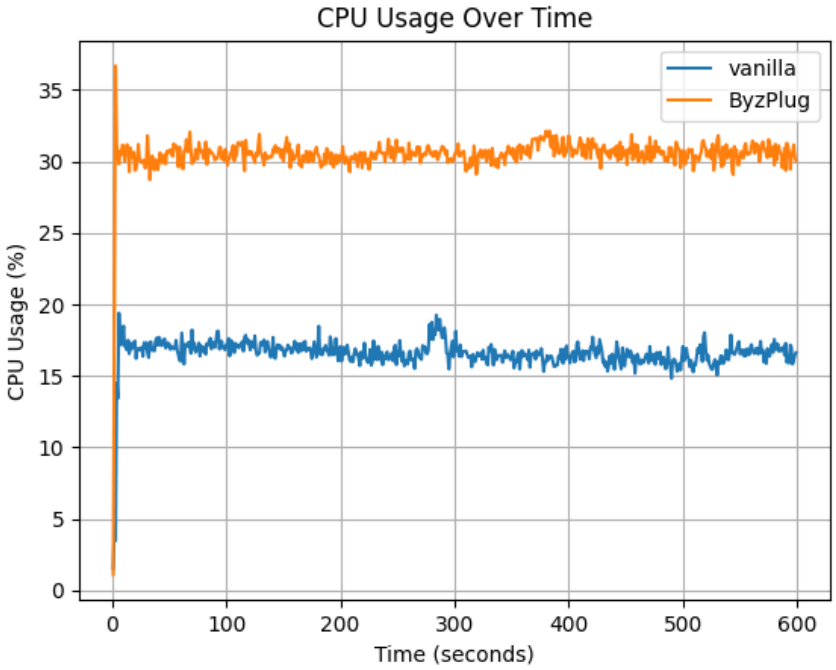

Figure 5.6: CPU load with vanilla setup and intercepting messages at the sender and receiver side.

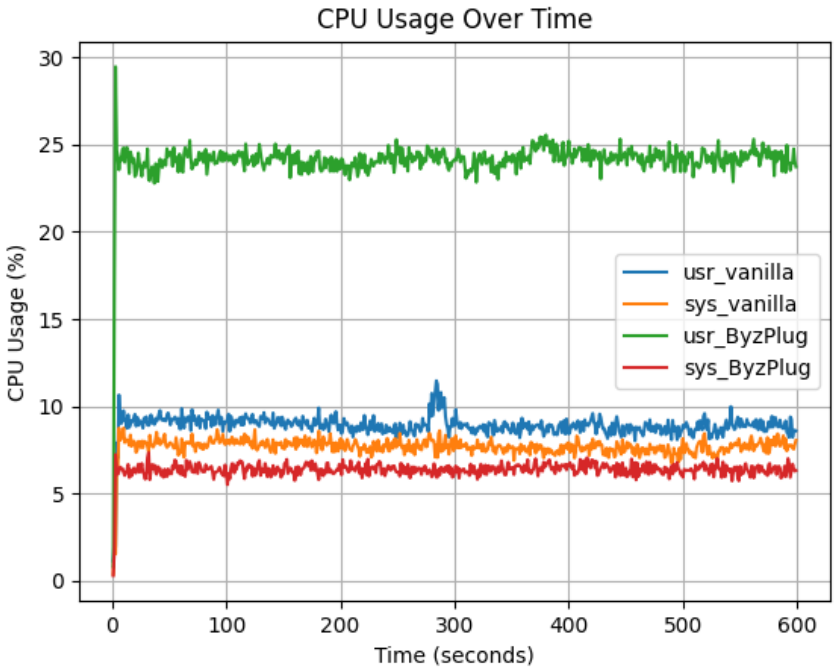

Figure 5.7: User and System CPU differentiated load with vanilla setup and intercepting messages at the sender and receiver side.

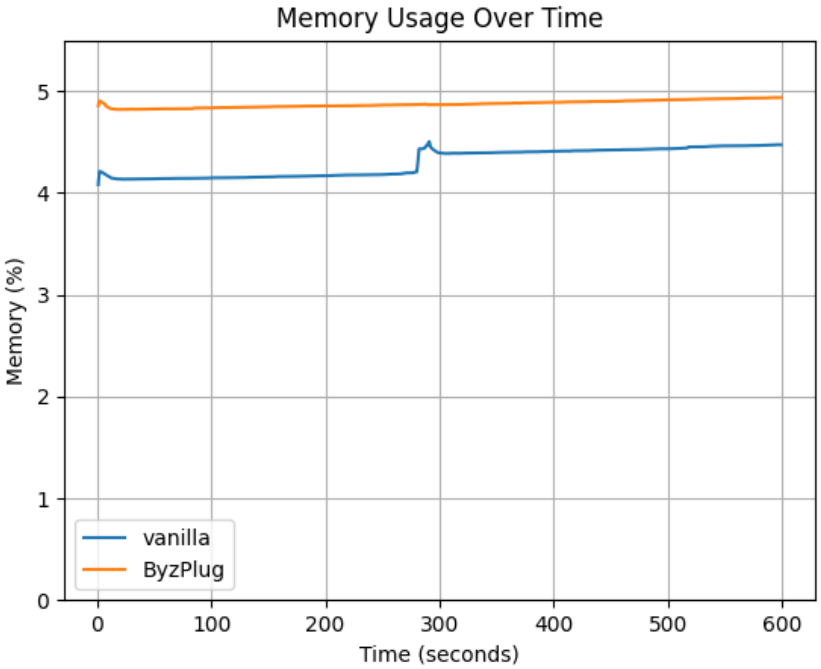

Figure 5.8: Memory usage with vanilla setup and intercepting messages at the sender and receiver side.

6

Conclusion

Contents

In this chapter, we present the final remarks of our research. We wrap up this thesis by giving a conclusion on the ideas presented, and discussing what we achieved. Finally we give insights on future work that could be done to improve our work.

6.1 Conclusion

In this work, we present ByzPlug, a protocol-agnostic fault injection tool that intercepts and manipulates all UDP/IP and TCP/IP process-level communication of a distributed system, without requiring any source code extensions or modifications. We discuss in Section 3.4 how this is a novel approach for testing the reliability of implementations of BFT protocols.

As its name suggests, testing a new protocol is as simple as ”plugging” in the fault injector to the same network in which the protocol nodes are running. In technical terms, this means deploying the necessary programs mentioned in Section 4.1 to its hook points, so that ByzPlug can intercept and

manipulate network traffic between the nodes.

ByzPlug gives the user the ability to manipulate and inject faults based on the protocol-specific information intercepted in packets. Doing so requires setting up a gRPC server that handles the packet manipulator `RPC` call defined in Algorithm 2.

During this work, we reproduced a safety violation in an implementation of HotStuff, proving that ByzPlug can be used to reproduce bugs in real systems. Section 5.1.1 describes how to reproduce this safety violation.

In Section 5.2 we evaluated the performance overhead of using ByzPlug to intercept and deserialize all messages exchanged in a distributed system. With 4 nodes running on the same virtual machine, the results are an (aggregated) overhead of 1.76x in CPU usage and 1.12x in memory usage. This results show that ByzPlug can also be used as a tool that enables full observability over process-level communication on a distributed system.

We conclude this work by saying that our objectives were met. The final result is ByzPlug, a fault injection tool that can be used in any BFT protocol without source code extensions or modifications, to reliably test its implementation. It can be used to find and reproduce bugs in BFT protocols, or as a tool that enables full observability over process-level communication in a network.

6.2 Future Work

We leave as future work some features that would enable us to automatically find bugs and have the necessary feedback from the tool to reproduce the found bugs. These consist of: *i*) automatically verifying the properties of consensus; *ii*) automatically generating the test cases to test the protocol; and *iii*) automatically generating an execution trace that can be fed back as a test case to reproduce a bug found.

Recall that all of these objectives should be achieved while maintaining the protocol-agnostic characteristic of ByzPlug. It is possible with our current tool to achieve all of *(i)*, *(ii)* and *(iii)*, to a specific protocol we want to test. However, if testing a new protocol all of this would have to be re-implemented to that specific protocol.

While *(i)* can be solved by using eBPF to track protocol decisions, *(ii)* and *(iii)* are not trivially solved. Every protocol uses different serialization and deserialization libraries. Moreover, these are written in different programming languages, and the structure of their messages differs completely from protocol to protocol. We argue that finding and implementing a working abstraction of the different protocol messages is another entire research work. Achieving progress in that would be a necessary step to improve ByzPlug to automatically find bugs in protocols, while still being protocol-agnostic.

Bibliography

[1] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: https://doi.org/10.1145/3281411.3281443

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[5] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, jul 1982. [Online]. Available: https://doi.org/10.1145/357172.357176

[6] D. Association, "Diembft," https://github.com/diem/diem, 2022.

[7] I. Abraham, G. Gueta, and D. Malkhi, "Hot-stuff the linear, optimal-resilience, one-message BFT devil," *CoRR*, vol. abs/1803.05069, 2018. [Online]. Available: http://arxiv.org/abs/1803.05069

[8] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," Cryptology ePrint Archive, Paper 2019/270, 2019, https://eprint.iacr.org/2019/270. [Online]. Available: https://eprint.iacr.org/2019/270

[9] M. M. Jalalzai, J. Niu, and C. Feng, "Fast-hotstuff: A fast and resilient hotstuff protocol," *CoRR*, vol. abs/2010.11454, 2020. [Online]. Available: https://arxiv.org/abs/2010.11454

[10] J. Kwon and E. Buchman, "Cosmos," https://v1.cosmos.network/resources/whitepaper, 2018.

[11] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[12] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, p. 398–461, nov 2002. [Online]. Available: https://doi.org/10.1145/571637.571640

[13] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Brief Announcement: Twins – BFT Systems Made Robust," in *35th International Symposium on Distributed Computing (DISC 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Gilbert, Ed., vol. 209, Dagstuhl, Germany, 2021, pp. 46:1–46:4.

[14] C. Drăgoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, "Testing consensus implementations using communication closure," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428278

[15] L. N. Winter, F. Buse, D. de Graaf, K. von Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586053

[16] D. Gupta, L. Perronne, and S. Bouchenak, "Bft-bench: Towards a practical evaluation of robustness and effectiveness of bft protocols," in *Distributed Applications and Interoperable Systems: 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings 16*. Springer, 2016, pp. 115–128.

[17] P.-L. Wang, T.-W. Chao, C.-C. Wu, and H.-C. Hsiao, "Tool: An efficient and flexible simulator for byzantine fault-tolerant protocols," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 287–294.

[18] J. Soares, R. Fernandez, M. Silva, T. Freitas, and R. Martins, "Zermia - a fault injector framework for testing byzantine fault tolerant protocols," in *Network and System Security*, M. Yang, C. Chen, and Y. Liu, Eds. Cham: Springer International Publishing, 2021, pp. 38–60.

[19] A. Starovoitov and D. Borkmann, "ebpf: Extended berkeley packet filter," https://ebpf.io/, 2024.

[20] D. M. Maofan Yin, "libhotstuff: A general-purpose bft state machine replication library with modularity and simplicity," https://github.com/hot-stuff/libhotstuff, 2018.

[21] Google, "Protocol buffers: A language-neutral, platform-neutral extensible mechanism for serializing structured data," https://github.com/protocolbuffers/protobuf, 2008, version 3.0, Available at: https://protobuf.dev/.

[22] Google, “grpc: A high performance, open-source universal rpc framework,” https://github.com/grpc/grpc, 2016, version 1.0, Available at: https://grpc.io/.

[23] Linux Kernel Contributors, “Af_xdp: Address family optimized for high-performance packet processing,” https://www.kernel.org/doc/html/latest/networking/af_xdp.html.

[24] Linux Foundation, “Linux kernel networking stack,” https://www.kernel.org/doc/html/latest/networking/index.html, 2023, version 6.0, Available at: https://www.kernel.org/.

[25] J. Postel, “Transmission control protocol,” Request for Comments 793, Internet Engineering Task Force, 1981, available at: https://www.rfc-editor.org/rfc/rfc793. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt

[26] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[27] DPDK Project, “Data plane development kit (dpdk),” https://www.dpdk.org, 2023, version 23.03, Available at: https://www.dpdk.org/.

[28] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, “Fcatch: Automatically detecting time-of-fault bugs in cloud systems,” *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 419–431, 2018.

[29] Google, “Kubernetes (k8s),” https://github.com/kubernetes/kubernetes, 2024.

[30] I. T. W. R. Center, “Wala,” https://github.com/wala/WALA.

[31] S. Chiba, “Javassist: Java bytecode engineering toolkit,” https://github.com/jboss-javassist/javassist, 1999.

[32] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, “Automatic reliability testing for cluster management controllers,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 143–159.

[33] L. Zhang, B. Morin, B. Baudry, and M. Monperrus, “Maximizing error injection realism for chaos engineering with system calls,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2695–2708, 2021.

[34] L. Zhang, J. Ron, B. Baudry, and M. Monperrus, “Chaos engineering of ethereum blockchain clients,” *Distributed Ledger Technologies: Research and Practice*, vol. 2, no. 3, pp. 1–18, 2023.

[35] T. go-ethereum Authors, “Goethereum,” https://github.com/ethereum/go-ethereum, 2013–2023.

[36] Nethermind, “Nethermind,” https://github.com/NethermindEth/nethermind, 2023.

[37] S. Sondhi, S. Saad, K. Shi, M. Mamun, and I. Traore, “Chaos engineering for understanding consensus algorithms performance in permissioned blockchains,” in *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 2021, pp. 51–59.

[38] P. Szilágy, “Clique proof-of-authority consensus protocol,” https://eips.ethereum.org/EIPS/eip-225, 2017.

[39] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC’14. USA: USENIX Association, 2014, p. 305–320.

[40] L. Holmberg, J. Heyman, and A. Baldwin, “Locust,” https://github.com/locustio/locust, 2011.

[41] A. Ledenev, “Pumba,” https://github.com/alexei-led/pumba, 2016.

[42] E. Foundation, “Aspectj,” https://github.com/eclipse-aspectj/aspectj, 2001.

[43] Diem, “Network playground,” https://github.com/diem/diem/blob/main/consensus/src/network_tests.rs.

[44] T. Yin, “https://github.com/determinant/salticidae,” https://github.com/Determinant/salticidae.