Karlsruhe Institute of Technology

Analysing Performance of Atomic Broadcast in the Asynchronous Setting

Master's Thesis of

Wen Zhan

at the Department of Informatics, Institute of Information Security and Dependability (KASTEL)
Decentralized Systems and Network Services Research Group

Reviewer:	Prof. Dr. Hannes Hartenstein
Second reviewer:	Prof. Dr. Martina Zitterbart
Advisor:	Marc Leinweber, M.Sc.

April 2022 – October 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Wen Zhan)

Abstract

The correctness of Atomic Broadcast to realize the State Machine Replication relies on the underlying theoretical model, with Byzantine fault-tolerance limiting the maximum faulty replica to less than 1/3 of the total amount and timing assumption determining the acceptable maximum message delay.

A stable network is fundamental to maintaining partially synchronous atomic broadcast functionality, so it may experience worse performance than the asynchronous one in the face of network influences and self-crashes when deployed in the real world. To determine whether an asynchronous atomic broadcast can outperform partially synchronous one, two algorithms, "Narwhal and Tusk" (briefly, Narwhal) and HotStuff, based on asynchrony and partial synchrony, respectively, are chosen as the objectives to discover how the external environment influences their behaviours.

Given the papers on Narwhal and HotStuff, both algorithms are analysed theoretically in detail to present how the protocol operates to reach consensus and what message complexity they can achieve in theory. Then, three real-world scenarios are specified for the purpose of emulating their behaviours: Best case, Fail-silent, and Communication delay. By leveraging their protocols implemented in Rust as the black box to observe atomic broadcast behaviours, a framework with Python scripts has been developed for practically benchmarking Narwhal and HotStuff in these three scenarios. To cancel the optimized impact of the batching technique on performance and ascertain the actual throughput of the protocol for consensus operation, both algorithms are configured to achieve a decision over only one transaction in each round of consensus.

Because the DAG-based Mempool supports paralleled transaction propagation, Narwhal also maintains more stable latency in the face of continuously growing input. Despite HotStuff's implementation of linear communication complexity, the leader remains a performance bottleneck, which leads to faster performance degradation in the face of growing replica sizes; In the Fail-silent case, Narwhal maintains its overall good performance as the number of failed replicas increases from zero to f. In contrast, HotStuff's performance gets much worse once a failed leader occurs, which interrupts the consensus processing and triggers `NEW-VIEW` phase, resulting in frequent leader elections. However, Narwhal suffers from a rapid drop in its TPS once the number of faulty replicas reaches an unknown threshold. When increasing communication delay, both algorithms suffer from TPS drop and latency increase due to excessive input transactions that overload the system. HotStuff stops reaching consensus once the delay exceeds the timer for detecting failures, while Narwhal still makes progress. We conclude that asynchronous Atomic Broadcast achieves better robustness due to its less dependency on external environment, provides larger scalability while maintains overall good performance, and tolerants longer message delay.

Zusammenfassung

Die Korrektheit von Atomic Broadcast zur Realisierung der State Machine Replication hängt von dem zugrunde liegenden theoretischen Modell ab, wobei die byzantinische Fehlertoleranz die maximale Anzahl fehlerhafter Replikate auf weniger als 1/3 der Gesamtmenge begrenzt und die Timing-Annahme die akzeptable maximale Nachrichtenverzögerung bestimmt.

Ein stabiles Netzwerk ist für die Aufrechterhaltung der teilweise synchronen atomaren Broadcast-Funktionalität von grundlegender Bedeutung, so dass bei einem Einsatz in der realen Welt angesichts von Netzwerkeinflüssen und Selbstabstürzen eine schlechtere Leistung als bei der asynchronen Variante auftreten kann. Um festzustellen, ob ein asynchroner atomarer Broadcast besser abschneiden kann als ein teilsynchroner, wurden zwei Algorithmen, "Narwhal and Tusk"(kurz Narwhal) und HotStuff, die auf Asynchronität bzw. Teilsynchronität beruhen, als Ziele ausgewählt, um herauszufinden, wie die externe Umgebung ihr Verhalten beeinflusst.

Anhand der Veröffentlichungen über Narwhal und HotStuff werden beide Algorithmen theoretisch im Detail analysiert, um darzustellen, wie das Protokoll funktioniert, um einen Konsens zu erreichen, und welche Nachrichtenkomplexität sie theoretisch erreichen können. Dann werden drei reale Szenarien spezifiziert, um ihr Verhalten zu emulieren: Bester Fall, Fail-silent und Kommunikationsverzögerung. Durch den Einsatz ihrer in Rust implementierten Protokolle als Blackbox zur Beobachtung des atomaren Broadcast-Verhaltens wurde ein Rahmen mit Python-Skripten entwickelt, um Narwhal und HotStuff in diesen drei Szenarien praktisch zu testen. Um die optimierte Auswirkung der Batching-Technik auf die Leistung aufzuheben und den tatsächlichen Durchsatz des Protokolls für den Konsensbetrieb zu ermitteln, werden beide Algorithmen so konfiguriert, dass in jeder Konsensrunde eine Entscheidung über nur eine Transaktion getroffen wird.

Da der DAG-basierte Mempool die parallele Weitergabe von Transaktionen unterstützt, bleibt die Latenzzeit von Narwhal auch bei kontinuierlich wachsenden Eingaben stabiler. Obwohl HotStuff eine lineare Kommunikationskomplexität implementiert hat, bleibt der Leader ein Leistungsengpass, was zu einem schnelleren Leistungsabfall angesichts wachsender Replikatgrößen führt; Im Fail-silent-Fall behält Narwhal seine insgesamt gute Leistung bei, wenn die Anzahl der ausgefallenen Replikate von Null auf f steigt. Im Gegensatz dazu verschlechtert sich die Leistung von HotStuff erheblich, sobald ein ausgefallener Leader auftritt, was die Konsensverarbeitung unterbricht und die `NEW-VIEW`-Phase auslöst, was zu häufigen Leader-Wahlen führt. Narwhal hingegen leidet unter einem schnellen Abfall seiner TPS, sobald die Anzahl der fehlerhaften Replikate einen unbekannten Schwellenwert erreicht. Wenn die Kommunikationsverzögerung zunimmt, leiden beide Algorithmen unter einem TPS-Abfall und einem Anstieg der Latenzzeit aufgrund von exzessiven Eingabetransaktionen, die das System überlasten. HotStuff erreicht keinen Konsens mehr, sobald die Verzögerung den Timer für die Erkennung von Fehlern über-

schreitet, während Narwhal weiterhin Fortschritte macht. Wir kommen zu dem Schluss, dass das asynchrone Atomic Broadcast aufgrund seiner geringeren Abhängigkeit von der externen Umgebung eine bessere Robustheit erreicht, eine größere Skalierbarkeit bei insgesamt guter Leistung bietet und längere Nachrichtenverzögerungen toleriert.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

A distributed system is a system whose components are spread over multiple computers, which coordinate their actions by exchanging messages in order to accomplish a common goal, such as reaching consensus on the system state. Overcoming the independent failure of components so that the entire system continues to function is a significant difficulty for distributed systems.

In such a system, the components located on different computers are called processes. A process can be a computer, a processor within a computer, or just a specific thread of execution within a processor [CGR11]. It might be faulty during execution, such as crashed down, or omitted a received message. All processes communicate with each other through network connection and collaborate as a group to realize the common goal. When a computer runs a process that is participating in a distributed system, it is referred to as a node.

In the field of distributed system, achieving overall system reliability in the presence of numerous failed processes is one of the fundamental problem, which requires coordinating correct processes to reach consensus. One scenario where consensus can be applied is how the bank's servers agree on which transactions are to be committed to the database in which order. Other real-world applications that require consensus include cloud computing, clock synchronization, blockchain, etc.

To realize a fault-tolerant distributed system, the technique *Atomic Broadcast* is rather helpful: An Atomic Broadcast, namely total-order broadcast, is a broadcast messaging protocol that guarantees that all messages are received reliably and in the same order by all correct participants of a distributed system [DSU04]. This type of broadcast is characterized as *atomic* because either every correct participant completes it successfully or all of them terminates it without experiencing any side effects. Atomic broadcast is quite important in the field of distributed system, which can be applied as the primitive for reaching consensus between all the correct participants.

The difficulty of implementing an Atomic Broadcast is affected by timing assumptions: Synchrony, partial synchrony, or full asynchrony. Atomic Broadcasts based on synchrony or partial synchrony have much stronger requirements for network reliability than those based on asynchrony. Asynchrony implies that messages can be delayed for an arbitrary but finite time span, i.e., any message will be delivered eventually at an unknown time point.

Implementing a deterministic consensus primitive in an asynchronous distributed system is, no matter of the assumed fault model, not possible because of the famous FLP impossibility [FLP85]. However, in recent years, different approaches have been published

that make asynchronous Atomic Broadcast a realistic paradigm. Examples are HoneyBadger BFT [Mil+16], Dumbo [Guo+20], DAG-Rider [Kei+21b], or Bullshark [Gir+22]. These projects motivate their focus on the asynchronous setting with improved robustness of Atomic Broadcast in wide-area networks, better performance, or less complex implementation.

The experimental results from these projects show that asynchronous Atomic Broadcasts outperform the partial synchronous approaches in which system will finally converge to be synchronous, e.g., classic PBFT [CL+99] and state-of-the-art HotStuff [Yin+18], with higher throughput and better scalability. In particular, the throughput of Dumbo and Bulldog can be scaled to some extent with the number of processes.

While the attack against PBFT presented in [Mil+16] demonstrate that synchronous BFT protocols can fail or suffer from performance degradation when network conditions are adversarial, how the adversarial network condition will affect the performance of asynchronous Atomic Broadcasts remains still unexplored.

In the actual world, process behaviour and network circumstances are unpredictable, and every type of faults is conceivable. Processes may crash down, messages can be delayed arbitrarily due to network congestions, causing the assumption of (partial) synchrony to fail. In a worse case, processes or network connection could be controlled by an adversarial, and these devices compromised by an attacker can collude to break the entire distributed system down. As an example, PBFT as described in [Mil+16] suffers from heavily performance degradation by constructing an adversarial network scheduler to continuously trigger view-change phase such that no more transaction could be processed.

The robustness of a distributed system is determined by how it responds to a certain class of faults. [Avi+04]. Unknown is how an asynchronous Atomic Broadcast behaves differently in comparison with a synchronous one when encountering various types of faults. However, the impact of certain type of faults on the system can be quantified by measuring the performance of an Atomic Broadcast, i.e., throughput, the number of requests that can be processed per second, and latency, the amount of time it takes for the client to receive the result after sending the request. These collected data allow for the quantification and evaluation of the performance differences between these two distinct types of Atomic Broadcasts.

In addition, it is unknown how the performance of asynchronous or partially synchronous Atomic Broadcast would be impacted by various types of faults, such as a continually increasing number of crashed processes, increased communication delay, or even intermittent network partitions.

The authors of HoneyBadger claimed [Mil+16] without providing empirical evidence that, even when the timing assumption is satisfied, partially synchronous protocols are slow to resume work after healing the partition of the network, whereas asynchronous protocols make progress as soon as messages are delivered. Further empirical research on the impact of the timing assumption and other faults on performance and robustness of an Atomic Broadcast algorithm is required.

Although, the work of Narwhal and Tusk [Dan+21] and the HotStuff [Yin+18] both presented the performance of these two types of Atomic Broadcast when facing faulty replicas, but they only concerned these fail-silent replicas which are not started at all from the beginning of the benchmarking. In other words, the real case they simulated is only a

smaller size of committee(2f+1 correct replicas participating in consensus, f faulty replicas never once correct and working). The real-world scenario is like all n = 3f+1 replicas are once correct and maintaining the consensus progress. But along with time, some of them might crash down and never resume working. This situation is more valuable to simulate to observe how an Atomic Broadcast behaves.

In contrast to the asynchronous situation, Atomic Broadcast may be implemented when (partial) synchrony [DLS88] is assumed by combining Reliable Broadcast with timers as fault detectors [CL+99]. Additionally, extant work has shown that performance and fault-tolerance can be increased significantly [Ver+11; DCK15; Liu+18] by introducing Trusted Execution Environments (TEEs) in (partial) synchronous setting at each node.

To ensure the security property, which implies that no two correct participants will make different decisions, a distributed system can tolerant up to f Byzantine faulty processes, with a total of $N = 3f + 1$ processes, as the cost to preclude so-called equivocation during reaching consensus [LSP82]. Equivocation is the act of dispersing contradictory information in the same context to different communication partners. TEEs can prevent the equivocation from being successful [Cle+12; Ver+11] which allows the Atomic Broadcast to tolerate up to $f < \frac{1}{2}n$ faulty processes. In the scenarios without TEEs, techniques such as PBFT must eliminate equivocation at the cost of a three-phase communication and the maximum number of $f < \frac{1}{3}n$ faulty processes.

Because timer is not existed as a fault detector in an asynchronous system, the correct processes collaborate differently in comparison with these in a (partially) synchronous one. HoneyBadger BFT and its descendants [Mil+16; Guo+20] fall back to asynchronous Binary Agreement with a common coin paradigm [MMR14] to reach consensus. The Binary Agreement schema necessitates a fairly strong notion of *validity*. More specifically, a decided input value must be proposed by a correct process. This form of validity is not achievable when there are more than $\frac{1}{3}n$ faulty processes [Xu21; LSP82] what cancels out most advantages from deploying TEEs.

Atomic Broadcast, on the other hand, does not necessarily require such strong validity. Known definitions merely demand that a decided value needs to be proposed by some process [PGS98; CL+99] or that a value proposed by a correct process will eventually be decided [Cri+95; Mil+16]. Bullshark [Gir+22] uses a Directed Acyclic Graph (DAG) and a common coin abstraction instead of a Binary Agreement, which may allow using TEEs in combination with a DAG to achieve Atomic Broadcast with more than $\frac{1}{3}n$ faulty processes.

1.2. Research questions

In recent years, there has been a lot of interest in Atomic Broadcasts for asynchronous or (partially) synchronous systems, and several near-proposed approaches, such as Tusk [Dan+21] or HotStuff [Yin+18], have achieved higher performance and robustness than classical approaches. Nonetheless, a systematic method for discovering the performance of these two distinct types of Atomic Broadcasts in facing a variety of scenarios is still absent.

The structure of an asynchronous atomic broadcast is more straightforward from an algorithmic design perspective, as there is no need to examine if the timing assumptions

are being broken. It is precisely for this reason that it is much more difficult to design an asynchronous Atomic Broadcast that is formally proven to be correct. Asynchronous Atomic broadcasts that have taken a lot of time and effort to design may turn out to be futile, unless we find that they are indeed exceptionally better than (partially) synchronous Atomic Broadcasts in some certain situations. Foremost, it must be determined whether there is necessarily a need for asynchronous atomic broadcast in some real-world scenarios.

For the purpose of determining the necessity for asynchronous Atomic Broadcasts, both forms of Atomic Broadcasts should be benchmarked to collect quantitative metrics, such as TPS and latency, to demonstrate the quantitative performance of the system. By analysing the performance indicators, it is possible to deduce that an asynchronous Atomic Broadcast may maintain good performance, but the (partially) synchronous approach suffers significant performance deterioration in some scenarios.

Although recent researches [Ver+11], [Zha+21] have demonstrated that TEEs can be implemented for Blockchain-based techniques to considerably improve performance and fault-tolerance, it has not yet been corroborated whether TEEs can be applied to an Atomic Broadcast based on the DAG structure.

To sum up, the following questions are unanswered:

1. Is asynchronous Atomic Broadcast necessarily required in some real-world scenarios?

2. Does an asynchronous Atomic Broadcast outperform the (partially) synchronous algorithm while ensuring better robustness?

3. Is a DAG-based Atomic Broadcast suited to take advantages of TEEs as it does not rely on Binary Agreement?

Approaches

A thoroughgoing literature review of asynchronous and (partially) synchronous atomic broadcasts is a necessary first step in answering unresolved research questions. Typical scenarios of atomic broadcasting applications will be summarized for experimental modelling of performance benchmarks during this phase.

Using a thematic reading approach, a comparison will be conducted between asynchronous and synchronous atomic broadcast in order to better comprehend the author's perspective and the complexities of implementing the atomic broadcasting algorithm.

In order to investigate the behaviour of algorithms in the face of crash faults, different timing models and network conditions, a representative synchronous algorithm and a representative asynchronous algorithm should be selected as the objects of an evaluation framework that is implemented to understand the robustness and performance of these two Atomic Broadcast.

Two of the most recent atomic broadcasting technologies were selected as evaluation targets for four scenarios: Narwhal, based on an asynchronous system, and HotStuff, based on a partially synchronous system, respectively.

Finally, the notions of validity must be strictly formally classified to help analyse the possibilities when using DAGs and TEEs in combination in asynchronous setting.

1.3. Outline

The thesis' remaining chapters are organized as followed:

Chapter 2 We introduce the theoretical background, system models of distributed system, and the definition of consensus.

Chapter 3 We present the related works on asynchronous consensus and the performance analysing of atomic broadcasts.

Chapter 4 We describe in detail the design idea of Narwhal mempool, and its properties, and reveal how Tusk implements random consensus on top of Narwhal.

Chapter 5 We carefully introduce HotStuff's three-phase commit model, analyse the linear complexity of its communication, and demonstrate the workflow of the HotStuff protocol through pseudocode.

Chapter 6 We present the framework and process setup of the experiment and design three scenarios to test the real responses of the two algorithms.

Chapter 7 We analysed the experimental results from Chapter 6, comparing and analysing the different behaviours of the two algorithms and the potential causes behind them, as well as making possible guesses for the unexplained behaviours. Also, possible directions for future work are given for issues that are still worth exploring.

Chapter 8 We summarized the most important concepts and findings of this thesis and commented on the practicality of the two atomic broadcasts.

2. Background

Lamport, Shostak, and Pease were the pioneers in introducing the Byzantine Generals Problem [LSP82] as a fundamental model for how to reach agreement on a value in a distributed system. A distributed system [AW04] is a type of computing system in which components may fail and there is incomplete information regarding whether a component has failed. Multiple processes that comprise a distributed system collaborate to achieve a common objective. They communicate with each other through network connections and cooperate as a group. In a decentralized architecture, these processes are deployed and operated by different entities. When participating in consensus, these processes controlled by different entities may crash down, or encounter network failures, or even behave maliciously to disrupt the consensus.

This chapter will introduce the fundamental concepts and theoretical models for solving consensus problems in a distributed system.

2.1. State Machine Replication

The State Machine Replication as introduced in [Sch90] is the most prevalent paradigm for the implementation of a distributed fault-tolerant system, in which a deterministic state machine is replicated to a group of servers and functions as a single state machine despite the failures of some servers. A fault-tolerance system consists of one or more servers which, as a whole, can execute the operations requested by the client. To provide fault-tolerance service to clients, servers of a distributed system can be replicated as state machines, and the State Machine Replication approach is utilized to coordinate the interactions between clients and servers.

Although a centralized server is the simplest approach to construct a service, the consequent service cannot survive the single-point-of-failure, which is unacceptable for a fault-tolerant system. For the purpose of designing fault-tolerant systems, state machine replication method can offer a wide range of applications. Without the requirement for centralized control, it can disguise faults to clients and simplify coordination between clients and multiple servers. The computation is replicated to physically isolated servers when the state machine replication approach is utilized to build a fault-tolerance system. Voting on the outputs generated by these independent replicas makes it possible to conceal the consequence of faults.

A state machine that stores the state of a system accepts a collection of requests (also known as commands or transactions) generated by a client as the input to the system and utilizes a transition function to sequentially apply these requests to generate an output and update the system's state. Requests are processed by a state machine one at a time and the transition function must ensure that the execution of a request, such as the execution

of a command or the processing of a transaction, is completed in a deterministic manner, so that a replica in its present state will be transferred to a deterministic state specified by the processed request and transition function.

The State Machine Replication approach guarantees that all correct replicas will process client requests atomically, which means either all correct replicas execute a request successfully or all replicas terminate the execution without bringing forth any side effects. To guarantee that all correct replicas complete the same state update, these requests will be applied to the system's state transitions on each correct replica in the same order. Two abstract concepts are defined in [Sch90] as the desirable properties of State Machine Replication:

- **Agreement**: Every non-faulty replica of the state machine receives every request.
- **Order**: Requests are processed in the same order receives every request.

How to guarantee these two properties is the most important aspect of implementing State Machine Replication. In order to resolve this issue, the technique *Atomic Broadcast* will be explained in details in section 2.5.

For formally describing the SMR approach, it is necessary to introduce the fundamental concepts in this subject area first. A state machine is composed of *state variables*, a set of state sets **S** and two sets used to specify the ranges of inputs and outputs: **I** and, **O** respectively. And the initial state of the system is distinguished as the state $Init \in \mathbf{S}$. The transition function δ for deterministically updating the system state is defined as:

$$\delta : Input \times State \longrightarrow Output \times State',$$

where $Input \in \mathbf{I}, Output \in \mathbf{O}$, and $State, State' \in \mathbf{S}$.

The defining attribute of a state machine is described in [Sch90] is the fact that it specifies a deterministic computation that reads the request stream, processes each request in turn, generates output, and transitions the current state of the system. This attribute is defined as **Semantic Characterization of State Machine** by Fred B. Schneider in [Sch90]:

> *Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.*

In the client-server model, all replicas starting at the same initial state *Init* simulate a state machine to build a distributed faults-tolerance system. And the following describes the coordination between clients and replicas:

1. A client asks for service by sending its requests to one of the replicas as if there is only single one state machine and waits for the replies from the replicated servers.
2. Any replica receiving the requests from a client will disperse these requests as the inputs to all other replicas.
3. A certain consensus mechanism has been specified and been deployed to guarantee these inputs will be executed the same order on all correct replicas and a same output is generated as the reply for client.

4. After the execution of an input, the state of each correct replica is transitioned to a new common one.

When a replica's behaviour no longer corresponds to its specification, it becomes faulty to the rest of the system's replicas. It is possible to build an f fault-tolerant state machine for a distributed system by reduplicating it and running a replica on each processor. Assuming each replica operating on a non-faulty processor begins in the same initial state *Init* and executes the same inputs in the same order, each will perform the same operation and provide the same output.

2.1.1. Byzantine Generals Problem

The term "Byzantine Generals Problem" is coined to describe a situation in which the participants of a distributed system must agree on a consistent strategy to prevent catastrophic collapse of the entire system while some of the participants are unreliable. A reliable distributed system must be capable of surviving the failure of one or more of its components. A malfunctioning component may exhibit such behaviour by yielding inconsistent or contradicting information to the different system components.

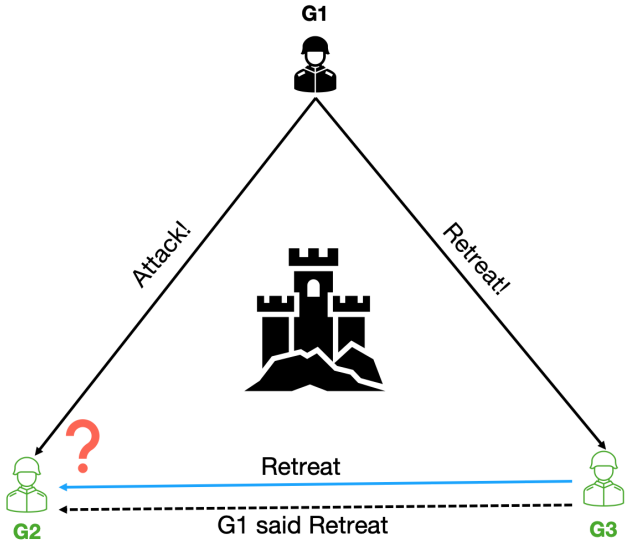


Figure 2.1.: The Byzantine General problem proves that if there exist f arbitrary faulty nodes, the total number of nodes must satisfy $n \geq 3f + 1$.

The Byzantine Generals Problem is an abstract representation of the difficulty of coping with this type of failure. Assume that the Byzantine army is camped outside the hostile city, with different divisions under the command of different generals. The generals are unable to directly contact one another and must rely on messengers to relay messages. They need to come up with a coordinated strategy and take the consistent action after assessing the adversary situation. Nevertheless, some of the generals could be traitors who are seeking to sabotage the agreement between the loyal generals.

It is essential for the loyal generals to have a strategy in place to ensure that

1. All the loyal generals must decide to take the same action.

2. A few traitors cannot induce the loyal generals to take an improper action.

The solution for the Byzantine Generals Problem appears to be straightforward, but an unexpected theorem proven in [LSP82] reflects its difficulty: no strategy will succeed until more than two-thirds of the generals remain loyal. In the scenario of distributed systems, given f the number of Byzantine faulty participants, then the total number of participants must be $n >= 3f + 1$.

Currently, the term "Byzantine fault" refers to an arbitrary fault, such as crashes, fails to send or receive a message when expected, or a malicious deviation from protocol (e.g., equivocation). Byzantine fault-tolerance is the characteristic of a distributed system that achieves to tolerant a maximum of f Byzantine-faulty processes while the remaining correct processes can still establish consensus.

The principal properties of Byzantine fault-tolerance protocols are characterized by the concepts of *safety* and *liveness*. In general, safety property specifies that the algorithm should not behave improperly. The property liveness of a distributed system promises that it will eventually respond to requests from clients who require for specific services. For a distributed system that guarantees properties of safety and liveness, a minimum of $3f + 1$ processes is required in order to tolerate f Byzantine faulty processes.

Byzantine failures are often the most catastrophic, it is rather possible to occur when the attacker can gain sufficient economical incentive by disrupting the system. $3f + 1$ replicas are necessary to survive from Byzantine failures when building an f fault-tolerance system based on the SMR approach.

In the State Machine Replication, the failure of one replica can be detected whenever its state or output deviates from the other replicas. A system failure occurs when no supermajority (at least $2f + 1$) of replicas have the same output, or when less than the supermajority of replicas return an output. Then, the output replied to the client should be *Failure*.

2.1.2. Cryptographic primitives

Several basic cryptography concepts are presented briefly in this section.

Cryptography provides the foundation for authentication of the message, as well as the confidentiality, and integrity. Two commonly used cryptographic primitives are essential requirements for exchanging message in a distributed system: one-way hash function and digital signature.

A hash function is a mathematical procedure that transforms data of arbitrary size (often referred to as the "message") to a fixed-size bit array (the "hash value", or "message digest"). The hash function is known as a one-way function, since it is nearly impossible to invert or reverse its computation. The only method to identify a message that generates a certain hash is to explore all possible inputs using brute force. A hash function must be deterministic, meaning that identical messages will always generate identical hash values.

A digital signature is a mathematical schema for authenticating digital information, such as messages or documents. A valid digital signature gives the recipient a very high

level of confidence that the message was created by a known sender (authenticity) and that the message has not been modified during transmission (integrity). Digital signatures utilize asymmetric cryptography. The private key holder signs the hash value of a message, anyone can verify that signature with the corresponding public key. A signer cannot successfully deny any message signed by its private key (non-repudiation).

The binding between the public key and the real identity of the entity is completed by the Public Key Infrastructure (PKI). When a certificate is signed by a trusted certificate authority, the holder can use the certificate's public key to establish secure communications with other parties.

Hash-based Message Authentication Code (HMAC) is a type of Message Authentication Code (MAC) involving a hash function and a secret key. It can be deployed to verify the data integrity and validate the authenticity of a message

2.2. Fault Model and Fault-Tolerance

A distributed system is composed of several processes, each of which is supposed to follow the specification and execute the algorithm assigned to it, and deal with various forms of failures. Supposing a static group consisting of n processes, and each process has a unique identification $\{p_1, p_2, \ldots p_n\}$, e.g., each is certified by the Public Key Infrastructure (PKI), such as X.509, and all processes know each other before starting their collaboration. These n processes are running on separated servers operated and controlled by different entities, and a process is correct if it functions properly and strictly adheres to the algorithm specification determined in advance.

A failure occurs whenever a process does not behave according to the protocol, such as a machine crashed or the network connection failed. In the worst-case scenario, computationally constrained adversaries are able to corrupt processes and conspire with other entities in order to disrupt the agreement between correct processes. By assumption, Byzantine processes are incapable to defeat cryptographic techniques such as asymmetric/symmetric encryption schema, digital signatures or hash-based message authentication code (HMAC) due to the computational constraints.

These adversary-corrupted processes can misbehave arbitrarily, which is defined as Byzantine faulty in [LSP82]. This type of fault is also referred to as arbitrarily faulty. To implement a Byzantine fault-tolerant distributed system, at most f processes can be arbitrarily faulty or be controlled by the adversary. For the purpose of theoretically analysing the safety and liveness properties of a distributed system, it is necessary to formally define the adversary's capabilities using the fault model. It demonstrates the malicious behaviours an adversary can accomplish on the processes it controls.

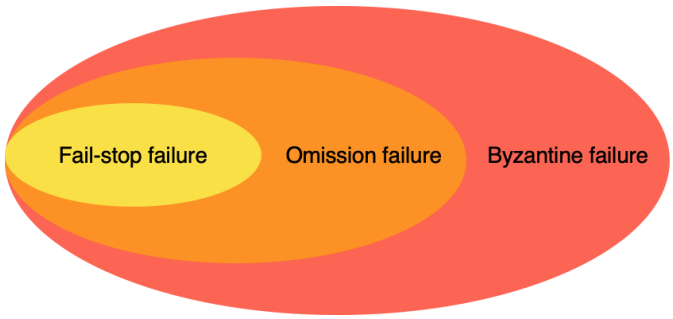


Figure 2.2.: The nested relationship of faults

- **Fail-silent failure**[Sch84]: A process operates correctly until it unexpectedly stops performing the algorithm and cannot recover.
- **Omission failure**: A process omits operations during communications, such as sending or receiving messages.
- **Byzantine failure**: A process can deviate arbitrarily from the protocol or even behaves maliciously to disrupt the system.

These four classes [DSU04] of faults are nested in the above order, and the Byzantine failure is almost the weakest assumption on the effects of a failure.

It is necessary to emphasize once again that cryptographic technologies are assumed invulnerable to adversaries limited to polynomial time computing power. The adversary also lacks knowledge of the correct process's secret key, hence it can neither impersonate the correct process, nor can tamper with messages from the correct processes without being detected.

2.3. Network and Timing Assumptions

Each pair of process is assumed to be connected by a secure, authenticated peer-to-peer channel that does not drop messages. Even though the adversary can completely control over the delivery schedule of messages transmitted between the correct processes, it cannot keep the messages from being delivered indefinitely. In addition to the failures caused by processes, the network, i.e., the communication between processes, does certainly conduct an impact on how well a BFT protocol performs, also needs to be formally analysed. The communication model for devices inside a network can be categorized as either synchrony or asynchrony. The concept of *Partial Synchrony* is further developed for pragmatic purposes to bridge the space between two ends of the spectrum for timing assumption.

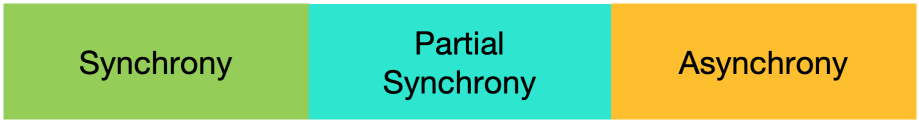


Figure 2.3.: The spectrum of three possible timing assumptions

Time-related matters are always challenging to handle in a distributed system. Before knowing what exactly partial synchrony is, it is necessary to define synchrony and asynchrony formally.

For a synchronous system, the upper bound Δ on the transfer time of a message delivered from one process to another across a network connection is fixed and known to all participants. For any message sent, the adversary can delay its delivery by at most Δ. In addition, there exists a known fixed upper bound Φ on the relative computation speed of any two processes, ensuring that any message may be processed within a given time period. Because the high-performance computers are ubiquitous nowadays, making the relative computation speed approximately to 1 and the communication delay an even bigger priority. Only the upper bound Δ for message delivery will be discussed in the following section.

By establishing a timer to calculate the communication latency, the correct processes in a synchronous system can determine whether another process is faulty or not. If a process does respond with the correct actions it is obligated to take within a prefixed delay limit Δ, it is correct. The adversary can delay the delivery of any message between correct processes by at most Δ until its behaviour is detected by the deployed timer.

In the asynchronous model, processes do not require physical clocks and timers, and each correct process can directly make progresses without detecting if timing violation or fail-silent happened. Due to the absence of a timeout mechanism, a process that has been waiting for a message that has not arrived cannot determine whether the message was delayed by the network congestion or if the sender had a problem and failed to send it at all. Therefore, if a process is awaiting messages from n processes and is aware that no more than f of them might be corrupted, it must take action after receiving $n - f$ messages rather than waiting for the messages from the remaining f processes.

Because the deterministic Byzantine fault-tolerance protocols are ruled out in asynchronous systems according to the well-known FLP impossibility [FLP85], a deterministic protocol must make certain stronger timing assumption. The best state of a distributed system is a fully synchronous system; nevertheless, this type of system is rather difficult to construct in the real-world. Partial synchrony is established as a more practical timing assumption for distributed systems. It is assumed that there exists some finite time bound Δ for message delay and a finite time span called Global Stabilization Time (GST). In the literature, partial synchrony can refer to two versions, which are theoretically equivalent:

- In the unknown bound model, there exists a fixed upper bound Δ, but it is not known to all participants beforehand, the protocol cannot use this bound as a parameter.

- In the eventually synchronous model, the bound Δ is known to all, but it is only guaranteed to hold at some unknown time T when the event GST happened.

Informally, the system behaves asynchronously until the end of GST and performs synchronously after time point T, such that any message sent at time t must be delivered by time $\Delta + max(t, T)$.

2.4. Consensus and Atomic Broadcast

Before diving into the Atomic Broadcast, it is important to understand the **consensus problem**. Reaching consensus is one of the most fundamental problems in distributed computing. This problem is about how multiple processors in a distributed system can agree on a common value. In other words, Consensus is the problem of *Agreement* between multiple processes on the value that one or more of those processes have proposed. Lamport has proposed two variants for the consensus problem. One is the interactive consistency model [PSL80], in which all processes need to calculate a vector of values and every element in the vector represents the value of a corresponding process; The other is Byzantine Generals Problem [LSP82] which assumes that there is a distinguished process proposes a value and others need to make an agreement on that value.

Given a distributed system comprises P of n processors linked via peer-to-peer communication channels. Each processor maintains a local state, which is updated over time based on the messages exchanged according to the algorithm specification. Processors try to reach consensus over the global system state by communicating with each other. In one round of communication, a processor sends a message to others and gathers messages sent by other processors in that round. Each massage m is identified by its round number r and the sender's signature. All processors should modify their local state in response to these exchanged messages and produce an output value. A correct process can decide what message it should send in a given round and how it updates its local state by conforming to the consensus protocol.

Reaching consensus can be broken down into two procedures as described on page 204 in [CGR11]: propose and decide:

1. Each process **proposes** an initial value v for consensus via broadcasting a message.
2. All correct processes must **decide** on a common value after exchanging messages.

In order to guarantee correctness, any consensus protocol must satisfy the following four properties defined in [CGR11]

- **Termination**: Each correct process eventually decides a value.
- **Validity**: If a process decides a value v, then v was proposed by some process.
- **Integrity**: No process decides twice in a round.
- **Agreement**: No two correct processes decide differently.

The consensus problem is in fact an abstraction of the problem of value-decision, and the central problem in implementing the SMR is precisely how to decide which operation (value) should be performed. At each local state updating, the SMR must ensure that all correct replicas execute the identical operation to main the same global system state. An existent technique to solve this problem is named as Atomic Broadcast, which enables processes to reliably broadcast messages in the same sequence. In a system consisting of multiple processes, all the correct processes can receive the same set of messages in the same order by deploying Atomic Broadcast. The term "atomic" implies that either all the

correct participants eventually transform into a common state after one instantiation of the atomic broadcasting, or none of them make any changes. The Atomic Broadcast can guarantee that the most recent updates to replicated data maintained by a group of process are consistently delivered to all group members, regardless of random communication delays or any type of failures.

To define the Atomic Broadcast, we need first formalize the schema for communication during reaching consensus: A process k can broadcast a message m by invoking a method $ABC\_send_k(m, r)$ in one communication round $r \in \mathbb{N}$ which is the sequence number of a message. Any process p_i received the message m when it outputs an event $ABC\_delivered_i(m, r, k)$.

Similar, the Atomic Broadcast layer is actual an abstraction of communication [Kei+21a] which provides these following four properties for the consensus protocol:

- **Validity**: If a correct process p_k called $ABC\_send_k(m, r)$, then any other correct process p_i will eventually output $ABC\_delivered_i(m, r, k)$ with probability 1.
- **Integrity**: Any correct process p_i can only output $ABC\_delivered_i(m, r, k)$ at most once for each $r \in \mathbb{N}$, regardless of m.
- **Agreement**: If a correct process p_i outputs an event $ABC\_delivered_i(m, r, k)$, then any other correct process p_j will eventually output $ABC\_delivered_j(m, r, k)$ with probability 1.
- **Total order**: If a correct process p_i outputs $ABC\_delivered_i(m, r, k)$ before $ABC\_delivered_i(m', r', k')$, then no other correct process p_j outputs $ABC\_delivered_j(m', r', k')$ without first outputting $ABC\_delivered_j(m, r, k)$.

In a real context of State Machine Replication, such as Ethereum, Atomic Broadcast is applicable to separate sequencing transactions from executing them. Transactions are totally ordered first, then an execution engine will validate them before applying them for the update of State Machine Replication.

Atomic Broadcast can be implemented in a variety of forms to solve the consensus problem in different situations. The fault model and the timing assumption are the most important characteristics we should consider when developing an Atomic Broadcast algorithm. Both have been introduced in the above part of this chapter.

2.5. The Performance of Atomic Broadcast

For the purpose of comparing different atomic broadcasts' behaviours, it is necessary to define some metrics for quantifying the performances of an Atomic Broadcast algorithm. The performance is measured in two dimensions: Throughput and response time. These metrics are defined as followed:

- **Throughput** refers to the number of successfully committed transactions per second, abbreviated as *TPS*. This metric directly reflects the transaction processing capability of the Atomic Broadcast.

- **Latency** indicates the time span between a transaction sent by the client and the result received from the server. This response time quantifies how long clients should wait before knowing the results of their transactions processing.

2.6. Data Structure

In SMR, the sequence of requests (in the field of Distributed Ledger Technology it is call *transactions*) executed by replicas can be recorded using two primary types of data structures. A request serves as an input to the SMR and trigger an update of the system update on every correct replica after deterministically executing that request. Blockchain stores requests in blocks, while DAG stores requests in nodes. In this section, these two data structures are explained and compared in details.

Blockchain, which is termed by Satoshi Nakamoto in the paper on Bitcoin [Nak09], consists of ordered units called *blocks*, in each block a header and a list of requests are included. In addition to other metadata, each block (except the very first one, called Genesis block) refers to a predecessor by keeping the hash value of its predecessor in the header area. In the ideal case, the blockchain has only one branch, i.e., each block on the chain has only one successor, and all blocks are arranged in strict chronological order. In order to ensure that requests are processed in the same order, all participants in the consensus maintain collaboratively a blockchain. Through some method, a leader is selected to take on the duty of adding a new block to the chain, which in turn drives an update to the system's state. However, a blockchain can be forked by adversaries to break protocol properties, which means that more than one block refers to the same predecessor.

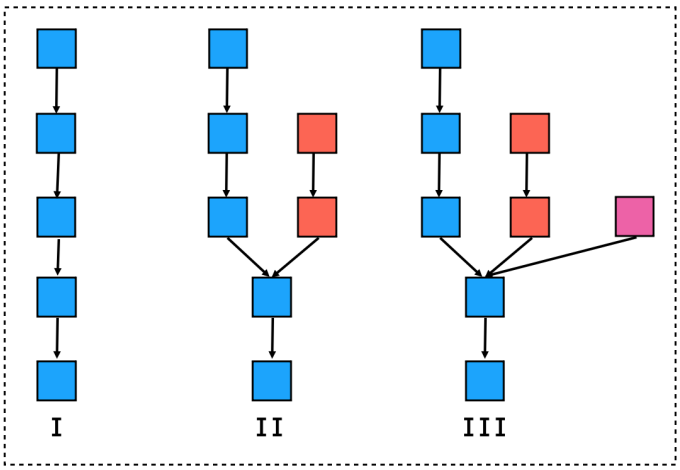


Figure 2.4.: Blockchain Unforked vs. Forked Possibilities

Some replicas may execute blocks on the blue branch, while others execute the red branch, resulting in inconsistent status updates for SMR(see fig 2.4). Blockchain forks may cause more serious problems in some financial applications, such as double spending. The Atomic Broadcast HotStuff for SMR introduced in Chapter 5 is based on blockchain structure

Directed Acyclic Graph (DAG) stores requests in it *vertex*, which has a similar structure as a block but contains multiple references to its predecessors.

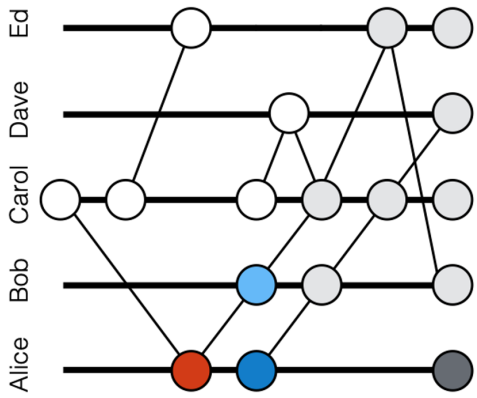


Figure 2.5.: Hashgraph

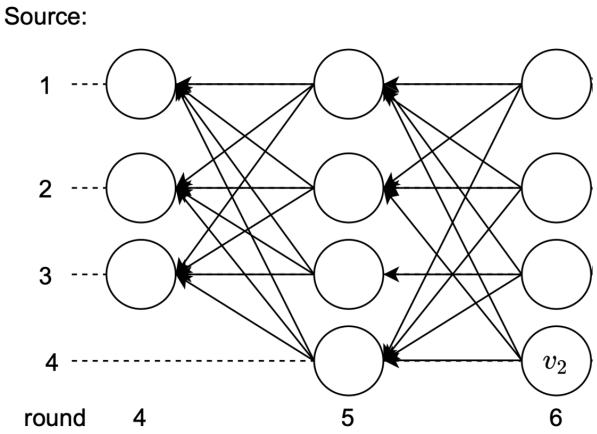


Figure 2.6.: DAG-Rider

Hashgraph is the first system to employ a DAG to address a consensus problem; it does this by generating an unstructured DAG (figre 2.5) in which each participant generates a chain for itself and each vertex in its chain has multiple refers to the vertices received from others to form a DAG. However, due to its irregular structured DAG, it is difficult to understand the correctness of the Hashgraph. So, Karl Crary verified the correctness of the Hashgraph formally in the paper[Cra21].

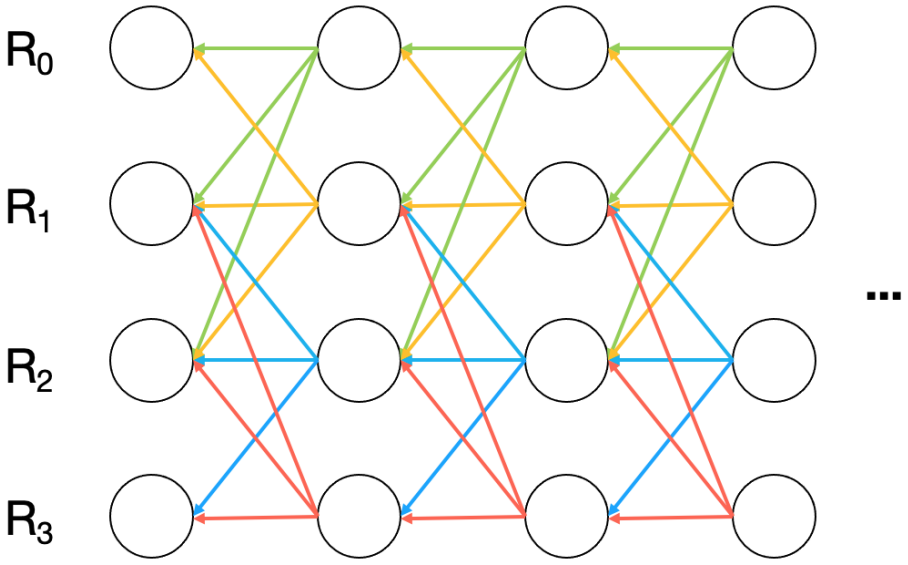


Figure 2.7.: Well-structured round-based DAG

The work on DAG-Rider proposes a round-based DAG (fig 2.6), makes it easy to understand the application of DAG wit Atomic Broadcast in the asynchronous case. The authors

of DAG-Rider have built a DAG, in which each vertex is assigned an integer, called the round number. All nodes labelled with the same number belong to a common round.

The Atomic Broadcast "Narwhal and Tusk" later proposed by the same authors is the objective of Chapter 4 in this thesis.

Blockchains and directed acyclic graphs (DAGs) are both abstraction layers used to record the communication between all participants, and as long as each correct participant ends up with the same blockchain or DAG, then their state updates will be eventually the same.

2.7. Selection of Atomic Broadcast

In this thesis, we aim to investigate various theoretical models and assumptions for atomic broadcasting, as well as their underlying principles and practical performance.

First, we decided to select a purely asynchronous atomic broadcast and a partially synchronous atomic broadcast as the objectives for this thesis, depending on the time assumptions. Besides the timing assumption, we anticipate two distinct atomic broadcasts, one based on the Directed Acyclic Graph (DAG) and the other on the Blockchain. Based on our literature review, we can say that DAGs are mostly used for asynchronous atomic broadcast, while Blockchain is mostly used for partially synchronous atomic broadcast. Therefore, we will analyse and contrast the two types of atomic broadcasts, one based on the DAG and the other on the Blockchain.

Hashgraph[Bai16], the first DAG-based atomic broadcasting technique, is a totally asynchronous algorithm, but its DAG structure is highly irregular, making it hard to discern whether it is correct. An extensive proof of correctness is provided for the succeeding DAG-Rider, which specifies a regularly-structured round-based DAG. In a later paper, the same authors presented Narwhal and Tusk, an algorithm that uses a more condensed DAG structure to reach consensus faster than DAG-Rider. Therefore, we decided to use Narwhal and Tusk as one of the atomic broadcasts that would have been studied in this thesis.

Partially synchronous atomic broadcast has been studied in extreme detail, from the very first PBFT[CL02], to the more recent MirBFT[Sta+19], or the very recent HotStuff[Yin+18] behind the Libra project. These algorithms have all been implemented within various blockchain projects, and they are all based on the blockchain structure. Facebook's Libra initiative had attracted a lot of interest, and the atomic broadcast technology behind it, the HotStuff, had just achieved the first linear communication complexity since PBFT. As a result, we selected HotStuff to investigate as the second algorithm in this paper.

We will introduce Narwhal and HotStuff in detail in Chapters 4 and 5, respectively, and present the experimental environment and benchmarking scenarios prepared for them in Chapter 6, and analyse their performance and behaviour in Chapter 7.

3. Related Work

Up recently, some consensus algorithms based on Atomic Broadcast in asynchronous setting, e.g., HoneyBadger BFT [Mil+16], DAG-Rider [Kei+21b] or Narwhal [Dan+21] demonstrated the possibility of the asynchronous BFT protocols to achieve rather good performance over wide area networks(WAN). Also, Facebook's HotStuff algorithm [Yin+18], which is a leader-based Byzantine fault-tolerant replication protocol for the partially synchronous model, achieved linear communication complexity and comparable throughput and latency to that of BFT-SMaRt[BSA14]. The research work in [Cra20] assessed four asynchronous atomic broadcasts based on Binary Agreement in a variety of experimental scenarios and demonstrated that each algorithm has situations where it performs better than all other algorithms in terms of specified metrics. However, comparative study on the performance of asynchronous and (partially) synchronous Atomic Broadcasts in various scenarios is still missing. Therefore, the main work of this thesis is to continue to design experimental scenarios based on this to verify and compare the performance and behavior of two different atomic broadcasts.

Atomic Broadcasts based on various implementations produce different levels of performance. Although the evaluation for HoneyBadger BFT showed it can achieve higher TPS than the classical PBFT, it still suffers from the scalability problem, i.e., consensus TPS decreases rapidly as the number of nodes increases. Dumbo as an improvement by reducing number of asynchronous binary agreement instances outperforms over HoneyBadger BFT with higher TPS and lower latency. Meanwhile, Dumbo has better scalability than HoneyBadger with little performance degradation.

The authors of DAG-Rider later proposed a protocol called Narwhal and Tusk [Dan+21], where the DAG component was redesigned to reduce the communication overhead by sending the hash value of a vertex instead of the original vertex. And the proof of existence is guaranteed by the availability certificate, which is a collection of signatures from these processes who stored that vertex. This DAG-based Mempool is named as Narwhal, on top of which a consensus protocol Tusk is employed to achieve consensus by integrating a common coin schema, such as threshold signature.

The authors of the pater [DGV05] have analysed the ability of processes to reach a consensus decision in a minimal number of rounds of information exchange for Asynchronous Byzantine Consensus in the best case and directly derived a meaningful bound on the time complexity of implementing robust general services using several replicas coordinated through consensus.

A meaningful bound on the time complexity of implementing robust general services using several replicas coordinated through consensus has been directly derived by the paper's [DGV05] authors, who have analysed the ability of processes to reach a consensus decision in a minimal number of rounds of information exchange for Asynchronous Byzantine Consensus in the best case. They gave the algorithms that match the bound in

the beat case, but how the time complexity will be affected in worse case is not explained, such as fail-silent.

In this paper [NET21] on faults modelling, a probabilistic model to assess the robustness of BFT atomic broadcast in the face of dynamic link and crash failures is presented. They do a theoretical investigation into the connection between the probabilistic model of component failures and system performance, but they do not support their conclusion with any kind of actual experimental evidence. Thus, their research prompted us to conduct actual benchmarking to check how various atomic broadcasting techniques react to the simulated faults before their applications into the real-world.

4. Analysis of Narwhal and Tusk

In this thesis, two Atomic Broadcasts are selected as the objective to explore their behaviours in three experimental scenarios. They are *Narwhal and Tusk* [Dan+21] for purely asynchronous network and *HotStuff* [Yin+18] based on partially synchronous system.

In this chapter, the problem that the Narwhal and Tusk protocol is trying to solve is first pointed out, then a higher-level view about Narwhal and Tusk is demonstrated, and finally the design idea of Narwhal and Tusk will be explained.

4.1. Overview

In the publication [Dan+21], the entire algorithm is designated by the name "Narwhal and Tusk", which is consisted of two components: Narwhal is a Mempool protocol based on a DAG structure, which is dedicated to the reliable dissemination and storage of causal histories of transactions. Tusk is a zero-message overhead asynchronous protocol for the consensus layer that collaborates with Narwhal to achieve consensus despite the occurrence of some faults.

The best-effort gossip Mempool is embedded into existing blockchain projects, such as Bitcoin [Nak09], Libra [Bau+19], which causes the double-transmission issue: transactions are shared first by the Mempool, and then the miner/leader broadcasts it again in a block when reaching consensus. It is the goal of the Narwhal Mempool protocol to decouple the dissemination of transaction data from the establishment of consensus.

The primary motivation for developing the Narwhal Mempool protocol came from this hypothesis[Dan+21]:

> *A better Mempool, that reliably distributes transactions, is the key enabler of a high-performance ledger. It should be separated from the consensus protocol altogether, leaving consensus only the job of ordering small fixed-size references. This leads to an overall system through-put being largely unaffected by consensus throughput.*

Tusk, inherited from DAG-Rider [Kei+21a], is a fully asynchronous algorithm for consensus layer that works with Narwhal. The validators of Tusk maintain a Narwhal Mempool to disseminate transactions, and also insert additional information into each of their blocks to generate a global perfect coin to reach a random consensus. Tusk's validators maintain a Narwhal Mempool active for the purpose of spreading transactions, and also insert extra data in each of their blocks to create a global perfect coin for the purpose of reaching random consensus over the order of transactions. The implementation of such a coin depends on PKI and a secure threshold ($f + 1 - of - n$) signature schema to guarantee the safety property.

In addition to resolving the double-transmission problem, the Narwhal Mempool protocol is intended to be a portable component. It is compatible not only with Tusk in a fully

asynchronous network setting, but also may be integrated into other Atomic Broadcast, like the partially synchronous consensus protocol HotStuff.

The results from the experiments conducted by the designers seems to support their hypothesis: The TPS of Narwhal-HotStuff is double that of HotStuff, whereas the TPS of Tusk exceeds that of Narwhal-HotStuff. Thus, the authors conclude that the performance of a monolithic Atomic Broadcast suffers greatly because transaction dissemination is put on the critical path of consensus.

The next sections of this chapter will go deep into the implementation details of Narwhal Mempool and the consensus layer Tusk.

4.2. System model

As explained in Chapter 2, assuming a message-passing system with n participants and a computationally constrained adversary who can compromise up to $f < n/3$ participants. Those participants that have been compromised by the adversary are called Byzantine or faulty, whereas the remaining are honest or correct.

In order to model real networks, we assume that communication links between honest participants are asynchronous and eventually reliable, in which message delay is unbounded and number of lost message is restricted. In addition, cryptographic primitives are solid and useful instruments for protecting the integrity of information.

4.3. Narwhal Design: DAG-based Mempool Protocol

The Narwhal Mempool protocol is built on concepts from reliable broadcast [BT85] and reliable storage[Abd+05]. In terms of abstraction, Narwhal is a key-value storage layer of blocks that is accessible to all participants for reading/writing blocks of transactions and extracting partial orders on blocks. Participants jointly maintaining a Narwhal Mempool are able to utilize a short key to refer to the original value stored in the shared storage layer and persuade others that the original value will be accessible to anybody upon request.

The Narwhal Mempool employs a DAG structure based on rounds to realize its storage abstraction, which is described in detail in the Chapter 2. Here is the formal description of Narwhal Mempool protocol:

The block is the element for a DAG structure, a block b in round r includes a list of transactions as well as a list of references to preceding blocks in the previous round $r-1$. A block (value) can be referred to by its identifier (key), the unique cryptographic digest of its contents, $d = hash(b)$. Referencing between blocks indicates the causal "happened-before" relation, denoting by $b \rightarrow b'$. Conventionally, all transactions in a referred block b are considered to have occurred before all transactions in the block b'.

Four block-related operations for storage layer are supported by the Narwhal Mempool protocol:

- $write(d, b)$ saves a block b associated with its digest (*key*) d, an unforgeable ***certificate-of-availability*** $c(d)$ on digest d is returned after a successful write operation. The

certificate of availability $c(d)$ can be generated only if at least $2f + 1$ validators received the block b and add it into their locally DAG.

- $valid(d, c(d))$ can verify the validity of a certificate-of-availability $c(d)$ and return a boolean value.
- $read(d)$ operation is invoked to extract the block b after a successful operation $write(d, b)$.
- $read\_causal(d)$ can fetch a set B of blocks such that $\forall b' \in B, b' \rightarrow \cdots \rightarrow read(d)$, which means for each $b' \in B$, there is a transitive "happened before" relation with the block b.

These participants maintaining a Narwhal Mempool can easily to realize the reliable dissemination and storage of causal histories of transactions by utilizing these four operations. And the these following properties defined by the authors in [Dan+21] must be satisfied for guarding Narwhal's safety:

- **Integrity**: Given a certified digest d (means $c(d)$ exists), any two correct participants who successfully invoking the operation $read(d)$ must extract the same value (identical block b).
- **Block-Availability**: After a successful $write(d, b)$ operation by an honest participant, the invocation of $read(d)$ operation from an honest participant will eventually return the value b.
- **Containment**: Let B be the set returned by invoking $read\_causal(d)$ operation, then for each $b' \in B$, the returned set B' after invoking $read\_causal(b')$ must satisfy the relation: $B' \subseteq B$, which means set B contains a subset B'.
- **2/3-Causality**: The returned set B by a successful operation $read\_causul(d)$ must include more than $2/3$ of the blocks written successfully before $write(d, b)$ was successfully invoked.
- **1/2-Chain Quality**: more than $1/2$ of the blocks in the returned set B by a successful operation $read\_causul(d)$ must be written successfully by honest participants.

Integrating with Narwhal Mempool, the consensus layer only needs to reach consensus over a small-sized certificate-of-availability $c(d)$ for a block digest d and order these certificates in the same way. This is possible because the certificate-of-availability for a block is equivalent to the block itself, as promised by the **Integrity** and **Block-availability** properties, which is the basis for segregating data dissemination from consensus reaching.

Once the agreement on a block digest, e.g., after synchrony is resumed, all the causally ordered blocks created in the previous asynchronous period can be safely and totally ordered, which is guaranteed by the properties of **Containment** and **2/3-Causality**. Therefore, Narwhal tries to compensate the consensus layer (even a partially synchronous one) with work in the asynchronous period to achieve decent throughput. Due to its

Chain-Quality [GKL15] feature, Blockchains utilizing the Narwhal protocol can also resist censorship.

Besides these four properties, Narwhal offers scalability by utilizing multiple workers to process transactions, allowing throughput to grow linearly with the number of resources each participant has. This is an engineering problem that is beyond the scope of this thesis, and we assume that each participant can have only one worker for transaction processing.

4.3.1. Intuitions towards Narwhal

Having established that the Narwhal Mempool protocol is meant to eliminate double transmission, it is now demonstrated how this set of properties defined in section 4.2.1 will eventually serve as the blueprint for the development of Narwhal:

1. Each participant broadcasts blocks to the Mempool instead of transactions for the purpose of data dissemination, and the leader, who is responsible for proposing a value to reach consensus, proposes the digest $d = hash(b)$ of a selected block b, the **integrity** of the original block b is protected by the Mempool layer.

2. To achieve the availability $c(d)$ of block b, it is reliably broadcast to all participants through Mempool, thus generating a **certificate-of-availability** for the block b which consists of at least $2f + 1$ signatures from receivers, indicating that the block b is available for download upon request. Then, the leader will propose this short certificate $c(d)$ for reaching consensus, which refers to block b and proves its availability.

3. Each Mempool block contains a list of at least $2f + 1$ certificate-of-availability of blocks in the previous round, resulting in a certificate $c(d)$ not only refers to the block b, but also its causal history. Proposing a fixed-size certificate $c(d)$ is equivalent to adding an extension (a new block b) to a sequence of blocks extracted from the b's causal history ($read\_causal(b)$ to retrieve the **2/3-Causality**)

4. In order to avoid some fast participants producing numerous blocks at high speed and thus others having to download a lot of blocks, resulting in no more bandwidth to share their own blocks, **Chain-Quality** is achieved by limiting the block creation rate: Each block must be assigned with a round number $r \in N$ to be valid, and it must contain a quorum of at least $2f + 1$ certificate-of-availability for blocks in the previous round $r - 1$. $2f + 1$ certificates are the threshold to advance round number, and at least $f + 1$ of which were created by the honest participants. Therefore, a malicious participant cannot advance the number round of Mempool until at least correct $f + 1$ participants have completed the previous round.

4.3.2. Core Design

Having understood how Narwhal's properties are defined and the purpose of these properties, it is now time to delve into how exactly Narwhal Mempool guarantees these properties.

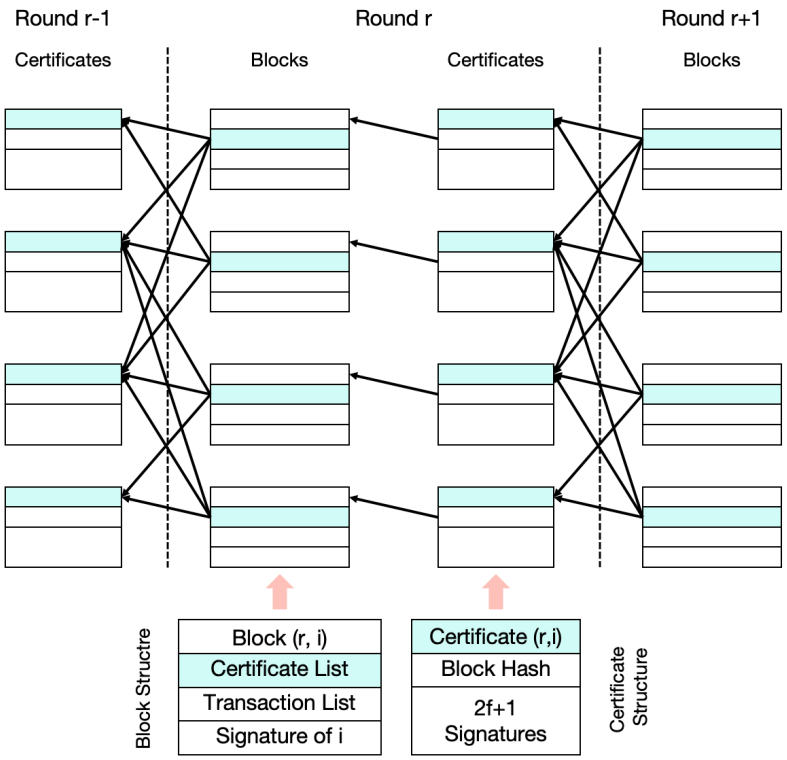


Figure 4.1.: Narwhal DAG's three-round structure: after collecting $2f + 1$ certificates for blocks in round $r - 1$, participants start creating new blocks for round r. Each valid block must contain $N - f$ certificates from round $r - 1$. Once a participant has collected $2f + 1$ signatures for the block it has created for round r to form the certificate, it will broadcast the certificate to other participants for inclusion in the blocks they will create for round $r + 1$.

As showed in figure 4.1, a participant p of Narwhal Mempool maintains a local round number r, starting at zero, and continuously receives transactions from its client, which are packaged into a transaction list to be included in the block b for round r. Meanwhile, p also continuously receives certificate-of-availability for blocks at round $r - 1$. As longs as $2f + 1$ certificates of blocks in round $r - 1$ created by distinct participants are gathered to form a list of certificates, which is included inside block b, then block b can be assigned with round number r, and signed by its creator p. After all of this effort, the creator of block b will broadcast it reliably [BT85] to ensure Integrity and Availability of this block and wait for acknowledgement, then advance the local round number to $r + 1$ for creating a new block. A correct participant can create at most one block for each round number.

Each block attached signature from its creator contains information about the creator's identity, a round number r, a list of transactions and a list of at least $2f + 1$ certificates for blocks in round $r - 1$. Any participant who receives a block must check its validity before saving it into the local DAG.

The validation must comply with the following criteria:

- A block must contain a valid signature generated by its creator,

- A block must have a same round number r as the participant checking it,
- A block must be either at round zero, or contain a list of at least $2f + 1$ certificates for blocks in round $r - 1$,
- A block must be the first received block from its creator for the round number r.

If a block passes the validation, then all correct participants should store it and send an acknowledgment to its creator by signing the block's digest, round number, creator's identity. As long as enough ($2f + 1$) acknowledgements for a block b are gathered by its creator, they will be combined to form a certificate of availability $c(d)$ for the block b, which includes the block digest $d = hash(b)$, round number, the creator's identity, and $2f + 1$ signatures from these participants who checked and stored block b. The Narwhal Mempool protocol initializes with round number $r = 0$, and all validators creates and verifies empty blocks for round 0 without any reference to certificates.

A certificate-of-availability of block b contains at least $2f + 1$ signatures from different participants, which means at least $f + 1$ of them are honest, and they have already checked and saved that block. As a result, block b can be retrieved at any moment by pull request. Quorum intersection of the $2f + 1$ certificates ensures properties of Block-Availability and Integrity for each block.

All blocks in the causal history are certified and promised available, satisfying causality, which can be proved by an inductive argument based on the fact that each valid block contains references to certificates from earlier rounds. The original work [Dan+21] contains more thorough security proofs for all properties of Narwhal.

Compared to previous consensus algorithms, Narwhal introduces the *garbage collection* feature that overcomes an important obstacle to the widespread adoption of DAG-based consensus algorithms, such as Hashgraph [Bai16]. This difficulty arises from the fact that a DAG is a local structure. It will eventually lead to all participants observing the same view of their local DAG, but it is impossible to predict when this will occur, causing all validators to maintain all blocks and certificates available for processing arbitrary old messages.

This obstacle is addressed by Narwhal's round-based message format. To validate a block in Narwhal, a validator doesn't have to look at the complete causal history, it can decide on the validity of the block only from information about the current round. Because each certified block carries enough information ($2f + 1$ references), its validity can be determined by referring to the cryptographic verification keys.

Narwhal utilizes the consensus layer to agree on the round number for the garbage collection, which is explained in the next section. In Narwhal, validators can run with a constant amount of memory. A participant needs $O(n)$ in-memory space to process blocks and certificates for the current round. Within a constant number of rounds (expectation: 4.5), all blocks will be eventually committed into the consensus output.

4.4. Tusk Design: Asynchronous Consensus Layer

Consensus algorithms operating in partial synchrony, such as Hotstuff, have a leader for proposing a block of transactions, who is selected by a deterministic leader election

schema. Tusk, however, the consensus layer designed to collaborate with Narwhal in a purely asynchronous setting (See the bottom half of figure 4.2), has to rely on a global perfect coin implementation to generate randomness for leader election, bypassing the FLP impossibility [FLP85].

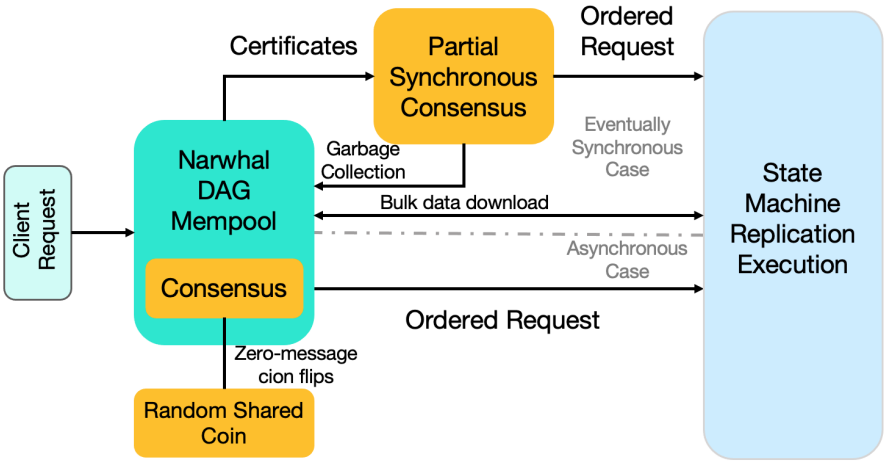


Figure 4.2.: Narwhal mempool can work with different types of consensus algorithms, both with asynchronous and partially synchronous algorithms.

Such a coin is unpredictable unless at least $f+1$ participants invoking a same instantiation of the coin, which takes an integer w as the input, and returns a random number to indicate a leader. In the paper on DAG-Rider [Kei+21a], the invocation of coin for an instantiation w by a participant p_i is denoted as $choose\_leader_i(w)$, returning a selected participant p_j as the leader for instantiation w. The global perfect coin has the following guarantees:

- **Agreement**: If two correct participants p_i and p_j have invoked the coin by calling $choose\_leader_i(w)$ and $choose\_leaer_j(w)$ with returned values p_1 and, p_2 respectively, then $p_1 = p_2$.
- **Termination**: If there are at least f+1 participants invoking the coin for w, then every $choose\_leader(w)$ call eventually returns a value.
- **Unpredictability**: As long as no more than f participant invoke the coin by calling $choose\_leader(w)$, then the return value is indistinguishable from random value except with negligible probability ϵ.

To implement a global perfect coin, the Public Key Infrastructure and a secure threshold signature schema with a threshold of $f + 1 - of - n$ are needed. Based on the certificate issued by the PKI, all participants can confirm each other's identities (the public keys) and then use threshold signature schema to achieve a common signature on a number w.

For the Agreement property to hold, all participants must agree on a single leader, and this is determined by the threshold signature value, which is revealed if any $f + 1$ participants executes the signature schema, e.g., the Practical threshold signatures [Sho00], and publish its result (the share). For unpredictability, the PKI must be relied upon to

guarantee that the adversary will be unable to generate enough shares to reveal randomness before a correct participant discloses its share. A trusted dealer is typically applied to generate and distribute keys for all participants, but this dealer is also avoidable by executing a protocol for Distributed Asynchronous Key Generation [KMS19] with $O(n^4)$ message complexity.

Beside operating a Narwhal Mempool as described in section 4.2.1, participants of Tusk consensus layer also append extra information into their blocks to generate a distributed global perfect coin for the consensus purpose: Leader election. After Narwhal constructs a causally ordered DAG, Tusk obtains it and completely orders its blocks with no extra communication required. In other words, the shared randomness is utilized by each participant to interpret its local perspective of the DAG and establish a global total order.

4.5. DAG Interpretation

In order to interpret the local DAG that Narwhal Mempool has constructed, a Tusk participant p_i must first divide its DAG_i into waves, where each wave consists of three successive rounds. For example, the very first wave of DAG_i contains three rounds: $DAG_i[1], DAG_i[2], DAG_i[3]$. For any wave $w \in \mathbb{N}$, its three rounds are indexed by $round(w, k)$, where $k \in [1, 2, 3]$. Formally, any round can be denoted as $DAG_i[round(w, k)]$, where $round(w, k) = 3(w - 1) + k$.

If the $DAG_i[r]$ contains at least $2f + 1$ blocks, then the round r is considered to be *completed* by the participant p_i. Participants must complete the current round before they can create or process blocks for the next round. And the wave w in DAG_i is completed if the participant completed $DAG_i[round(w, 3)]$

A wave w is completed as followed description:

1. Each participant create their blocks in the $round(w, 1)$ and broadcast it reliably to all other participants, meanwhile receiving and validating (as described in section 4.2.1.2) blocks created by others for the $round(w, 1)$.

2. After a participant gathering at least $2f + 1$ blocks (including the block created itself), it advances round number to $round(w, 2)$ and create the block for $round(w, 2)$, in which a list of at least $2f + 1$ certificates referring to the blocks in $round(w, 1)$ is included as "votes" to these referred blocks.

3. If a participant completed $round(w, 2)$, it will generate randomness by signing the wave number w with its private key. This signature will be added to the created block for $round(w, 3)$ to reveal the participant's share.

4. After a wave w is completed, at least $2f + 1$ blocks containing partial signatures in $round(w, 3)$ are collected to reconstruct the whole signature as an indicator of the leader in $round(w, 1)$.

Having mastered these fundamental ideas, it is now time to examine how to totally order a DAG as showed in figure 4.3.

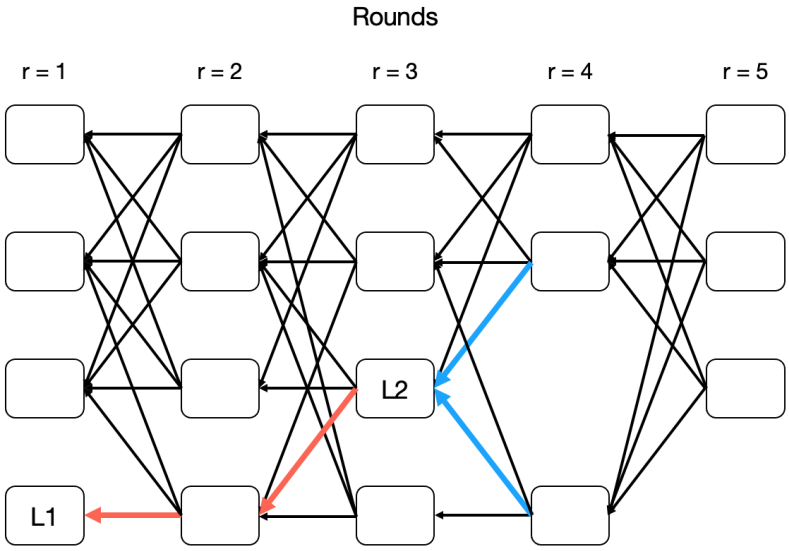


Figure 4.3.: Tusk protocol commit example: the odd-numbered rounds shown in the figure would perform a leadership election. Since the leader $L1$ elected in the first round contains less than f+1 support, as shown by the red line in the figure, it cannot be committed at that moment; whereas the leader $L2$ elected in the third round has enough support, as shown by the blue line in the figure, so L2 will be successfully committed

The basic idea is to interpret the DAG as a structure consisted of waves, and attempt to commit a single randomly selected leader block in the first round of each wave: $round(w, 1)$. Once the sequence of leaders is determined, all the blocks included in their causal histories will be ordered deterministically. The global perfect coin is invoked by calling $choose\_leader(w)$ to select a participant p_j randomly after the wave w is completed, and the block in $round(w, 1)$ created by p_j is then considered as the leader of the wave w.

Each correct participant p should commit the randomly selected leader block b of the wave w if there are $f + 1$ blocks in the second round of the wave w ($round(w, 2)$) that refer to b. Under this circumstance, a participant p then deterministically orders b's causal history up to the point of garbage collection, such as by calling the DFS algorithm.

To ensure that all participants are ordering block leaders in the same order, participants **recursively** commit leaders for waves obeying the following steps:

1. A participant sets the leader in the wave w as a candidate and push it into a stack S to be ordered.

2. Then, the participant finds the last committed wave w' and for each wave from w to w', p_i checks if there is a path from the current top candidate in stack S to the leader block b of wave i.

3. If such a path exists, then the leader of block b wave i will be pushed to stack S as a candidate to be ordered.

4. This procedure is executed recursively from wave $w-1$, ... to w', until a leader block in some wave i' ($w > i' \geq w'$) doesn't satisfy this path-condition.

5. Then, all candidates in the stack S will be ordered and committed by pop order, recursively ordering.

Once the order of leader blocks are determined, all the blocks in their causal histories are ordered by executing a deterministical algorithm. Then, a total order for local DAG is established according to randomness generated by the global perfect coin. The properties defined in previous sections will guarantee that all correct participant will eventually observe their local DAGs in the same view. all correct participants will eventually observe their local DAGs from the same perspective, according to the properties specified in previous sections. In DAG-Rider [Kei+21a], the safety argument for this mechanism has been proven formally with more details.

4.6. Implementation

The implementation of Narwhal and Tusk is open-sourced in Rust code and is available on GitHub[1], which use Tokio for asynchronous networks and ed25519-dalek for elliptic curve-based signatures.

In order to improve the performance of Narwhal and Tusk from an engineering point of view, the authors combined the two roles of the participants: worker and primary. A worker is responsible for receiving requests from clients and building a block, exchanging message with workers of other participants, passing the digest of a block to primary, who only handles the trust domain with its private key to generate a signature for a block or an acknowledgement. With this role differentiation, the processing and propagation of transactions can be separated from the consensus reaching to achieve better scalability and concurrency. A Primary can collaborate with multiple workers to improve the TPS of the system without significantly increasing the latency of the system.

To appropriately implement the distributed system abstractions, TCP is employed to provide reliable point-to-point communications. A queue of messages to be sent between peers is kept in memory and waits for send away to other peers over a persistent TCP channel. If the TCP channel drops, a re-establish will be tried to send the stored messages. Messages sent to unavailable peers are limited to a limited amount, since the primary or worker will eventually decide that they are no longer necessary for making progress and remove them from memory.

There are about 4,000 lines of code (LOC) in the implementation and another 2,000 LOC in the unit tests. The core protocols for Narwhal and Tusk are implemented in Rust, but all benchmarking scripts are in Python and executed via Fabric. In Chapter 6 and 7, how to benchmark the codebase and interpret the benchmarking results is described.

[1] https://github.com/asonnino/narwhal

5. Analysis of HotStuff

Following the presentation of the Narwhal and Tusk, another Atomic broadcast is examined closely in this chapter: HotStuff, which is an Atomic Broadcast implemented for Byzantine fault-tolerant replication protocol in the partially synchronous model. First, we give a brief overview of the problem with PBFT-like algorithms' scalability and then explain how HotStuff solves this problem while maintaining linear communication complexity. Then, the functionalities of HotStuff protocol are demonstrated by giving pseudocode.

5.1. Overview

In the context of State Machine Replication (SMR), the system as a whole provides a replicated service whose state is duplicated across n deterministic replicas. HotStuff is a BFT SMR technique used to guarantee that, despite the efforts of up to f Byzantine replicas, all correct replicas agree with the execution order of client's requests. This assures that $n - f$ non-faulty replicas execute requests identically and produce identical output for the same request. As introduced in Chapter 2, HotStuff guarantees liveness once network communication becomes synchronous after some unknown global stabilization time (GST). It empowers a correct leader to drive the protocol to consensus with communication complexity that is linear in the number of replicas after the communication delay resumes synchronous, but loses liveness property during the asynchronous periods.

Numerous BFT solutions are built on top of the practical aspect that correct leaders take their responsibility to drive consensus in two scenarios of message exchange. In the first scenario, the safety property of a consensus decision is guaranteed through the formation of a quorum certificate (QC) consisting of $(n - f)$ votes (the digital signature). For each QC, an integer will be stamped as the height indicator for the current view. In the second scenario, the liveness property is guaranteed by selecting a new correct leader for proposing the next consensus decision. This second scenario is called view-change, which is the epicentre of a BFT protocol based on partial synchronous setting. *View-change* is triggered after a faulty leader is detected, and each correct replica broadcasts a `NEW-VIEW` message to report its own known QC of consensus decision, which is stamped with a current highest view number.

Many BFT algorithms have been greatly hindered in their capacity to scale to a much larger n due to the scalability problem introduced by PBFT's $O(n^2)$ communication complexity. To drive the three-phase commit protocol, which is essential to ensuring the correctness of PBFT, all replicas must broadcast messages to all other replicas, squaring the complexity of communication.

HotStuff implements a three-phase core that allows only one leader to need to broadcast a message, but the other replicas reply to the leader in an unicast fashion, which achieves

a linear communication complexity with respect to the number of replica. For HotStuff, the cost of selecting a new leader to drive consensus is no more than the cost of a current leader, which makes frequent succession of leaders possible.

5.2. System Model

In analysing HotStuff, the system is considered as a fixed set of $n = 3f + 1$ replicas, indexed by $i \in [n] = \{1, 2, ..., n\}$, and up to f replicas are Byzantine faulty, the remaining ones are correct. Network communication is peer-to-peer, authenticated and reliable: one correct replica receives a message from another correct replica when and only when the latter sends that message to the former. When referring to a "broadcast", it involves a correct broadcaster sending the same point-to-point message to all replicas, including itself. The partial synchrony defined in Chapter 2 is adopted to model the communication delay: there is a known bound Δ and an unknown Global Stabilization Time such that after GST, all transmissions between correct replicas will be delivered within the time Δ.

HotStuff utilizes a secure threshold signature schema to guarantee its safety property. In a (k, n)-threshold signature schema, there is a single public key known to all replicas and used to verify the composed signature, and each of the n replicas holds a distinct private key to generate its partial signature.

The i-th replica can use its private key to contribute a partial signature $\rho_i \leftarrow tsign_i(m)$ on a message m. A set of partial signatures $\{\rho_i\}_{i \in I}$, where the size of a set I of replicas must satisfy $|I| \geq k$ and each $\rho_i \leftarrow tsign_i(m)$, can be composed to form a complete digital signature $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$ on the message m. Anyone can verify this signature σ using the function $tverify(m, \sigma)$ with the known public key. A secure digital threshold signature scheme can promise that if $\rho_i \leftarrow tsign_i(m)$ for each $i \in I, |I| \geq k$, and if $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$, then the verification $tverify(m, \sigma)$ must return **true**. The threshold in HotStuff is $k = 2f + 1$, and any adversary who corrupts only up to f replicas has negligible probability of producing a signature σ for the message m.

A cryptographic hash function h, which maps an arbitrary-length input to a fixed-length output, must be collision resistant, which informally requires that the probability of an adversary producing two different inputs m and m' such that $h(m) = h(m')$ is negligible. Therefore, an input m to the protocol can be identified by its hash value, $h(m)$.

5.3. Data Structures

The HotStuff protocol functions in a sequence of views, which are stamped with monotonically increasing integer numbers

In each view, there is a unique leader determined according to leader election criteria and known to all replicas, taking the responsibility of the leader. The authors of HotStuff used the terminology of tree to describe its local data structure, which is exactly the "blockchain" defined in Chapter 2. Each replicas stores a *tree* of requests as its local data structure, which is a blockchain with forks. Each fork is named a *branch* in the tree for HotStuff. Each *node* in the tree, which is the block in a blockchain, contains a proposed

batch of requests from the client, metadata associated with the protocol, and a parent link to a previous created node. A branch led by a certain node is the path from that node to the tree's root (the Genesis block) via visiting its parent link (see figure 5.1).

Along with the running of the HotStuff protocol, a monotonically growing branch of the executed clients' requests becomes to be committed as the record of consensus sequence. In order to commit a branch, the leader of a particular view proposing a node into that branch must accumulate enough votes from a quorum of replicas in three phases of message exchanging: `PREPARE`, `PRE-COMMIT`, `COMMIT`.

The key component in the HotStuff protocol is a collection of $(n - f)$ votes over a leader proposal, referred to as a Quorum Certificate (QC), which is a set of partial signatures generated over a particular node and a view number. By using $tcombine()$ utility, a QC can be combined to form a representation of $(n - f)$ signed votes as a single authenticator.

Before delving into a phase-by-phase description of the operations of the HotStuff protocol, four related data structure types are declared first, which are very basic concepts needed to understand the HotStuff protocol specification.

Message: A message m in HotStuff has a fixed set of fields that are populated using the MSG utility defined in Algorithm 1:

- $m.curView$: The current view number of a replica who sent this message m
- $m.type$: The type of message, five types are defined for exchanging messages in HotStuff protocol as {`NEW-VIEW, PREPARE, PRE-COMMIT, COMMIT, DECIDE`}
- $m.node$: A proposed node containing clients' requests as the object for consensus
- $m.justify$: An optional field inside m to carry the QC for different phases, a leader uses it broadcast $prepareQC$, $precommitQC$, or $commitQC$, and a replica uses it in `NEW-VIEW` message to carry the highest $prepareQC$ it has.
- $m.paritalSig$: The partial digital signature for the authentication of a message m signed by a replicas over the tuple $\langle m.type, m.viewNumber, m.node \rangle$, which is generated by applying the utility VoteMSG defined in Algorithm 1.

Quorum certificate: A QC is actually a collection of partial signatures over a tuple $\langle type, viewNumber, node \rangle$ signed by $(n - f)$ replicas for a message m. Given a QC qc, $qc.type$, $qc.viewNumber$ and $qc.node$ refer to the matching fields of the original tuple, respectively.

Tree and branch: A tree is the local data structure of each replica to record its commit history for executed requests. The root is the very first node, which is called "genesis block" in terms of blockchain. Every request is contained inside a node that has a parent link referring to the parent node. A replica delivers a message only after the branch led by the node is available in its local tree. In practice, a recipient who has fallen behind can catch up by retrieving missing nodes from other replicas. If none of the two branches is an extension of the other, they are *conflicting*. Two nodes are conflicting if the branches led by them are *conflicting*, as shown in figure 5.1.

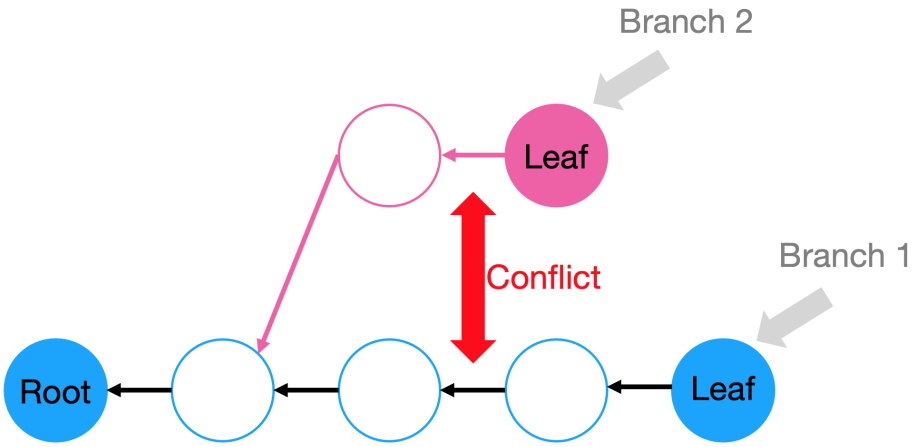


Figure 5.1.: HotStuff "Tree" structure: Two conflicting branches led by different leaf nodes are showed to illustrate a fork happened.

Bookkeeping variables: A replica uses additional local variables for tracking the system state:

- *viewNumber* is incremented either by committing a decision or by a `NEXT-VIEW` interrupt, the initial value is 1;
- *prepareQC* saves the highest QC for which a replicas voted `PRE-COMMIT` message, the initial value is $\perp$.
- *lockedQC* saves the highest QC for which a replica voted `COMMIT` message, the initial value is $\perp$;

The following algorithm 1 describes the data structure and its utilities precisely.

5.4. Algorithm Specification

There are multiple communication phases that occur between the leader and the replica during the consensus processing over a decision, as showed in figure 5.2 and algorithm 2. This section elaborates on each of these phases.

5.4.1. Phases of Communication

1. `PREPARE` phase. A new leader l, which is deterministically elected by $l = viewNumber \mod n$, assumes its responsibility after collecting `NEW-VIEW` messages from $(n - f)$ replicas for current view *viewNumber*. The `NEW-VIEW` message is sent by a replica when it encounters an NextView interrupt, such as it doesn't receive a proper message from the old leader. A `NEW-VIEW` message carries *prepareQC* stamped with the highest view number that a replica has received ($\perp$ if none), as defined in NextView utility. The leader selects a branch which has the highest *prepareQC* after processes these $(n - f)$ `NEW-VIEW` message to start next view. The leader picks the *prepareQC* with the highest view number in the

Algorithm 1 Utilities for replica r

function MSG($type, node, qc$)
    $m.type \leftarrow type$
    $m.viewNumber \leftarrow curView$
    $m.node \leftarrow node$
    $m.justify \leftarrow qc$
    return m
end function
function VoteMSG($type, node, qc$)
    $m \leftarrow$ MSG$(type, node, qc)$
    $m.partialSig \leftarrow tsign_r(\langle m.type, m.viewNumber, m.node\rangle)$
    return m
end function
function CreateLeaf($parent, request$)
    $b.parent \leftarrow parent$
    $b.request \leftarrow request$
    return b
end function
function QC(V)
    $qc.type \leftarrow m.type : m \in V$
    $qc.viewNumber \leftarrow m.viewNumber : m \in V$
    $qc.node \leftarrow m.node : m \in V$
    $qc.sig \leftarrow tcombine(\langle qc.type, qc.viewNumber, qc.node\rangle, \{m.partialSig | m \in V\})$
    return qc
end function
function MatchingMSG(m, t, v)
    return $(m.type = t) \wedge (m.viewNumber = v)$
end function
function MatchingQC(qc, t, v)
    return $(qc.type = t) \wedge (qc.viewNumber = v)$
end function
function SafeNode($node, qc$)
    return ($node$ extends from $lockedQC.node$) $\vee$ ▷ Safety rule
    ($qc.viewNumber > lockedQC.viewNumber$) ▷ Liveness rule
end function

$(n - f)$ NEW-VIEW messages, denoted as *highQC*, which confirms that there are no higher views to reach the commit decision to keep the branch led by this *highQC.node* safe.

The leader utilizes the CreateLeaf function to extend the branch led by the *highQC.node* with a new proposed node. When called by the leader, CreateLeaf generates a leaf node containing requests and inserts a digest of the parent node (*highQC.node*) into it. The returned leaf node is called *curProposal*. Then, this *curProposal* is broadcast to all other replicas through a PREPARE message. This message carries a *highQC* proving that all correct replicas are extending a common branch to guarantee the safety property. After receiving the PREPARE message for the current view from the leader l, a replica r utilizes the SafeNode method to determine whether to accept it.

The SafeNode method is the core ingredient for the security of HotStuff. It takes a message m as the input and checks its QC $m.justify$ to determine whether it is safe to accept *m.node*. For the purpose of the examination, two rules were adopted: The safety rule examines that a branch led by *m.node* must extend from the checker's current locked node *lockedQC.node*, which guarantees that the send and receiver are processing a common branch; The liveness rule guarantees that $m.justify$ must hold a higher view number than its current $lockedQC.viewNumber$ to avoid processing an old message. The invocation of this function is true as long as either one of two rules holds.

If SafeNode returns True, the replica r will send a PREPARE vote message with its partial signature (generated by deploying $tsign_r$) for the proposed node to the leader.

2. PRE-COMMIT phase. As long as the leader accumulates $(n - f)$ PREPARE votes for *curProposal*, it combines them into a *prepareQC* containing a complete signature σ collaboratively generated by $(n - f)$ replicas. This justification *prepareQC* is broadcast to all replicas through a PRE-COMMIT message. A correct replica responds to the leader with its PRE-COMMIT vote attached with a signature for the proposal.

3. COMMIT phase. This phase is similar to PRE-COMMIT phase. When the leader receives $(n - f)$ PRE-COMMIT votes, they will be combined to form a *precommitQC* and broadcast in a COMMIT message; replicas respond to the *precommitQC* with a COMMIT vote message. Crucially, at this point, the local *lockedQC* of a replica is set to the *precommitQC* it received, making the replicas locked on *precommitQC*.

4. DECIDE phase. After $(n - f)$ COMMIT votes are collected by the leader, it combines them into a *commitQC*. Then, it sends the *commitQC* in a DECIDE message to all other replicas. Upon receiving the DECIDE message, a replica considers the proposal embodied in the *commitQC* a committed decision, and executes the requests in the committed node. After execution, replicas increment their viewNumber to $curView + 1$ and starts the next view.

5. NextView interrupt. In all above phases, a replicas r only waits for a message from the leader at view $viewNumber$ for a timeout period. Once the replica doesn't receive the corresponding message within the timeout period, an auxiliary utility NextView(viewNumber) will interrupt the waiting, increase the $viewNumber$ to $viewNumber + 1$ and starts a new view by sending a NEW-VIEW message to the new leader $l = viewNumber + 1 \mod n$.

5.4.2. Algorithm Pseudocode

The specification in Algorithm 2 is described as an iterated view-by-view loop. In each view, a replica executes successive phases based on its role. As seen in the algorithm 2,

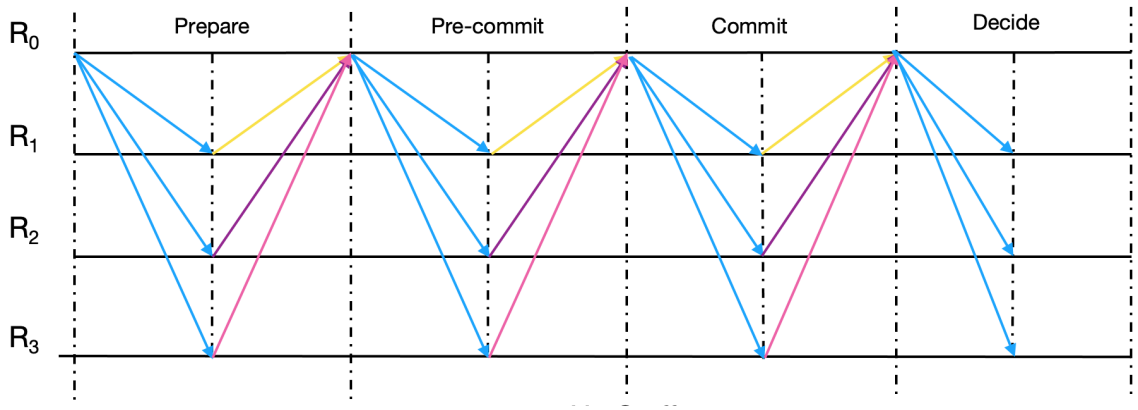


Figure 5.2.: Communication phases

the "as a replica" or "as a leader" block. The execution of an "as" block across replicas is proceeded concurrently and atomically. A NextView break can abort all operations in any "as" block and jump to the "Finally" block.

The correctness and complexity of the protocol Hotstuff have been proven formally in the work[Yin+18]. The authors also proposed a chained HotStuff to accelerate the consensus interval with pipeline design, which is beyond the scope of this thesis.

5.5. Implementation

The open repository[1] on GitHub provides a basic implementation of the HotStuff consensus protocol in Rust with but uses real cryptography dalek [2], network stack tokio [3], and storage utility rocksdb [4]. It is designed with a separation between mempool and consensus layer. A replica receives clients' transactions, batches them into payloads of the mempool and broadcasts them to other replicas; The purpose is to disseminate client's transactions to the other replicas before consensus layer attempts to sequence them (thus allowing a more egalitarian bandwidth utilization amongst replicas). This design choice also reduces client-perceived latency, as a replica does not necessarily need to wait to become leader to propose the transactions of its clients: other replicas may include their transactions after receiving them from the mempool batches.

Consensus is then reached on the hashes of these mempool batches after `DECIDE` messages are executed. Whenever the consensus layer needs to create a node proposal, it queries its local mempool for the hashes of one or many payloads to include in its node. Its query specifies a maximum number of bytes it wishes to receive (configurable) to ensure an upper bound on the size of its proposals. The mempool replies to the consensus layer with as many hashes of payloads as possible (while respecting the size constraint in the query), but may also reply with an empty vector in case there are no payloads available. This

[1] https://github.com/asonnino/hotstuff
[2] https://doc.dalek.rs/ed25519_dalek/
[3] https://docs.rs/tokio/latest/tokio/
[4] https://docs.rs/rocksdb/latest/rocksdb/

Algorithm 2 Basic HotStuff Protocol (for replica r)

for $curView \leftarrow 1, 2, 3, \ldots$ **do**
▷ PREPARE phase
as a leader
  wait for $(n - f)$ NEW-VIEW messages: $M \leftarrow \{m | \text{MATCHINGMSG}(m, \texttt{NEW-VIEW}, curVIEW - 1)\}$
  $highQC \leftarrow \left(\arg\max_{m \in M}\{m.justify.viewNumber\}\right).justify$
  $curProposal \leftarrow \text{CREATELEAF}(highQC.node, \text{client's requensts})$
  broadcast MSG(PREPARE, $curProposal$, $highQC$)
as a replica
  wait for message m: MATCHINGMSG(m, PREPARE, $curView$) from LEADER($curView$)
  if $m.node$ extends from $m.justify.node \wedge \text{SAFENODE}(m.node, m.justify)$ **then**
    send VOTEMSG(PREPARE, $m.node$, $\perp$) to LEADER($curView$)
  end if
▷ PRE-COMMIT phase
as a leader
  wait for $(n - f)$ votes: $V \leftarrow \{v | \text{MATCHINGMSG}(v, \texttt{PREPARE}, curView)\}$
  $prepareQC \leftarrow \text{QC}(V)$
  broadcast MSG(PRE-COMMIT, $\perp$, $prepareQC$)
as a replica
  wait for message m : MATCHINGQC($m.justify$, PREPARE, $curView$) from LEADER($curView$)
  $prepareQC \leftarrow m.justify$
  send VOTEMSG(PRE-COMMIT, $m.justify$, $.node$, $\perp$) to LEADER($curView$)
▷ COMMIT phase
as a leader
  wait for $(n - f)$ votes: $V \leftarrow \{v | \text{MATCHINGMSG}(v, \texttt{PRE-COMMIT}, curView)\}$
  $precommitQC \leftarrow \text{QC}(V)$
  broadcast MSG(COMMIT, $\perp$, $precommitQC$)
as a replica
  wait for message m : MATCHINGQC($m.justify$, PRE-COMMIT, $curView$) from LEADER($curView$)
  $lockedQC \leftarrow m.justify$
  send VOTEMSG(COMMIT, $m.justify.node$, $\perp$) to LEADER($curView$)
▷ DECIDE phase
as a leader
  wait for $(n - f)$ votes: $V \leftarrow \{v | \text{MATCHINGMSG}(v, \texttt{COMMIT}, curView)\}$
  $commitQC \leftarrow m.justify$
  broadcast MSG(DECIDE, $\perp$, $commitQC$)
as a replica
  wait for message m : MATCHINGQC($m.justify$, COMMIT, $curView$) from LEADER($curView$)
  execute request through $m.justify.node$, and respond to clients
▷ Finally
  NEXTVIEW interrupt: goto this line if NEXTVIEW($curView$) is called during "wait for" in any phase
  send MSG(NEW-VIEW, $\perp$, $prepareQC$) to LEADER($curView + 1$)
end for

may happen if you configure the batch size of mempool too high. Also, if the consensus' payload size is set too low, the mempool buffers will fill up with hashes of payloads waiting to be sequenced and eventually starts dropping client's transactions (to avoid running out of memory). The size of this mempool buffer is configurable through the mempool parameters.

6. Experimental Study

In this chapter, the construction of the test environment and the design of the experimental scenarios for the two types of atomic broadcasts are described in detail.

6.1. Experimental Framework

Both projects open-sourced on GitHub are maintained commonly by one of the authors: Alberto Sonnino. These two codebases are written in Rust, and all benchmarking scripts are written in Python and run with the library Fabric [1].

When running benchmarks, the codebase is automatically compiled with the feature flag `benchmark`. This enables the replicas to print some special log entries that can be read by the python scripts and used to calculate performance.

After cloning the repositories and installing all dependencies, Fabric tasks are utilized to run benchmarks on the local machine or on the remote server. In the original `fabfile.py`, there are two tasks defined for local model and remote model respectively. But this remote model is applied for the Amazon Web Service (AWS), which is quite expensive option to do our experiment. So, we extend the local task in the `fabfile.py` to be capable to run benchmarking on our own servers.

In the following description, the command `fab local` means to start benchmarking on the **ten servers** managed by the DSN research team. These ten servers are indexed as `mpc-0, mpc-1, ..., mpc-9`, each has 16 virtual CPUs supported by Intel Xeon E-2288G @ 3.70GHz with 32GB memory. The host system installed on each server is Linux Ubuntu server 20.04.

6.1.1. Configurations

Listing 6.1: Fabric task to start nodes on the server for benchmarking

```
...
@task
def local(ctx):
    LocalBench(bench_parameters, node_parameters).run()
    ...
```

The command `fab local` starts a fab task (defined as list 6.1) which takes two types of parameters as the configurations for benchmarking, the benchmark parameters and the nodes parameters. For Narwhal, they are specified as followed:

[1] https://docs.fabfile.org/en/stable/

Listing 6.2: Benchmark parameters

```
bench_params = {
    "nodes": 10,
    "workers": 1,
    "faults": 0,
    "rate": 4000,
    "tx_size": 512,
    "duration": 50
}
```

Listing 6.3: Node parameters

```
node_params = {
    "header_size": 100,
    "max_header_delay": 500,
    "gc_depth": 50,
    "sync_retry_delay":10000,
    "sync_retry_nodes": 10,
    "batch_size": 513,
    "max_batch_delay": 10000 }
```

In the list 6.2 and 6.3 shows the parameters for Narwhal benchmarking:

- `nodes`: The number of replicas participating in consensus, these replicas form a committee.
- `workers`: The number of workers per replicas to handle transactions, it's set as `1` in all experiments to get rid of the effect of optimization.
- `rate`: The input rate (tx/s) at which clients submit transactions to the workers
- `tx_size`: The size of each transaction created by client, in Bytes. Clients of the Narwhal protocol and the HotStuff protocol generate `512-byte` random strings as transactions.
- `faults`: The number of faulty replicas, up to f = `nodes//3`
- `duration`: The duration of benchmarking in seconds, each of our benchmarking lasts for `50` seconds.
- `header_size`: The predefined header size of each block, the primary creates a new header when it received enough $(2f + 1)$ parents and enough batches' digests to reach this size in bytes.
- `max_header_delay`: The maximum delay in milliseconds that the primary waits between creating two headers, even if the header did not reach the `header_size`.
- `batch_size`: The predefined batch size in Bytes. The workers seal a batch of transactions to create a block when it reaches this size. We should set this as **512 Bytes** to restrict that workers can only gather up to one transaction for each block. However, for an uncertain reason, this parameter will lead to great degradation (near to zero) of TPS unless we set it **513**.
- `max_batch_delay`: The maximum delay in milliseconds after which the workers seal a batch of transactions, even if the `max_batch_size` is not reached.
- `sync_retry_delay`: The delay after which the synchronizer sends `sync` requests, denominated in millisecond.
- `sync_retry_nodes`: Determine with how many nodes to sync when re-trying to send `sync` request. These nodes are picked at random from the committee.

- `gc_depth`: The depth of the garbage collection, denominated in number of rounds.

For HotStuff, they are specified as:

Listing 6.4: Benchmark parameters

```
bench_params = {



    "nodes": 10,
    "faults": 0,
    "rate": 4000,
    "tx_size": 512,
    "duration": 50



}
```

Listing 6.5: Node parameters

```
node_params = {
    "consensus": {
        "max_payload_size": 32,
        "min_block_delay": 0,
        "sync_retry_delay": 1000,
        "timeout_delay": 1000},
    "mempool": {
        "max_payload_size": 500000,
        "min_block_delay": 0,
        "sync_retry_delay": 1000,
        "queue_capacity": 100000}}
```

The `bench_params` for HotStuff benchmarking are similar with these for Narwhal, except there is no `workers`. For `node_params`, there is a different structure for HotStuff, which specifies parameters for `consensus` and `mempool` separately:

- `max_payload_size (consensus)`: The predefined size of payload inside a consensus message, we set it as **`32 Bytes`** to restrict that only the digest of one transaction will be contained during one round of consensus operation.
- `min_block_delay (consensus)`: The minimum delay a replica waits between creating two consensus blocks.
- `sync_retry_delay (consensus)`: Replicas re-broadcast `sync` requests when once timeout in milliseconds is reached.
- `timeout_delay`: Replicas trigger a `view-change` once this timeout in milliseconds reached.
- `max_payload_size (mempool)`: The predefined size of payload inside a mempool message, we set it to a large number to accelerate the transaction dissemination across replicas.
- `min_block_delay (mempool)`: The minimum delay a replica waits between sending two messages for transaction dissemination.
- `sync_retry_delay (mempool)`: Replicas re-broadcast `sync` requests when once timeout in milliseconds is reached.
- `queue_capacity`: The maximum number of batch digests that the mempool holds before dropping client transactions. To avoid running out of memory, it is usually set to a large number.

In addition to these two profiles `bench_paramters.json` and `node_parameters.json` for setting the benchmark, there is an important file `committee.json` for identification before

replication starting consensus. This `committee` file contains entries like the following lists 6.6 and 6.7 showed, it records the public key and communication address for each replica to help them exchange message directly through peer-to-peer network connection. Narwhal requires five ports per replica for message passing, but HotStuff only requires three ports per replica.

Listing 6.6: Narwhal Committee

```
{
    "authorities": {
        "+5QfEzuiu+
            DMwgWHI9RsKTQW7cRzGie20vyyZv3t1gg
            =": {
            "primary": {
                "primary_to_primary": "
                    129.13.88.180:9035",
                "worker_to_primary": "
                    129.13.88.180:9036"
            },
            "stake": 1,
            "workers": {
                "0": {
                    "primary_to_worker": "
                        129.13.88.180:9037",
                    "transactions": "
                        129.13.88.180:9038",
                    "worker_to_worker": "
                        129.13.88.180:9039"
                }
            }
        },
        ...
        ...
}
```

Listing 6.7: HotStuff Committee

```
{
    "consensus": {
        "authorities": {
            "CZksyvyrrNKIAJUCYbPbTLNsGoCJ1NPY+
                zAE2W/GQ0U=": {
                "address": "129.13.88.188:9000
                    ",
                "name": "
                    CZksyvyrrNKIAJUCYbPbTLNsGoCJ1NPY
                    +zAE2W/GQ0U=",
                "stake": 1
            },
            ...
    "mempool": {
        "authorities": {
            "CZksyvyrrNKIAJUCYbPbTLNsGoCJ1NPY+
                zAE2W/GQ0U=": {
                "front_address": "
                    129.13.88.188:9001",
                "mempool_address": "
                    129.13.88.188:9002",
                "name": "
                    CZksyvyrrNKIAJUCYbPbTLNsGoCJ1NPY
                    +zAE2W/GQ0U="
            },
            ...
}
```

For each replicas r, it owns a private file `replica-r.json` that stores its private key locally to sign each message it sends.

Listing 6.8: Narwhal Keyfile

```
{
  "name": "+5QfEzuiu+
      DMwgWHI9RsKTQW7cRzGie20vyyZv3t1gg=",
  "secret": "9g5EEHzyJPPh0tmqvJy4azK1VUwfdmTJk
      /1
      NluOGYBz7lB8TO6K74MzCBYcj1GwpNBbtxHMaJ7bS
      /LJm/e3WCA=="
}
```

Listing 6.9: HotStuff Keyfile

```
{
  "name": "CZksyvyrrNKIAJUCYbPbTLNsGoCJ1NPY+
      zAE2W/GQ0U=",
  "secret": "8EC27fsGfzheq/6
      ff0BMkgiD4LFEExhllgQFZIcM9z4JmSzK/
      Kus0ogAlQJhs9tMs2wagInU09j7MATZb8ZDRQ==
      "
}
```

6.1.2. Benchmarking Flow

The command `fab local` first recompiles the code in release mode with the benchmark feature flag activated to enable the replicas to write entries to log files that are later parsed by the fabric task `fab parsing` to calculate the performance. It then generates the configuration files and keys for each node, and runs the benchmarking with the specified parameters. Finally, it parses all generated log files and displays a summary of the benchmarking similarly to lists 6.10 and 6.11.

Listing 6.10: Narwhal Benchmarking Results

```
-----------------------------------------
 SUMMARY:
-----------------------------------------
 + CONFIG:
 Faults: 0 node(s)
 Committee size: 10 nodes
 Worker(s) per node: 1 workers
 Input rate: 4,000 tx/s
 Transaction size: 512 B
 Execution time: 50 s

 Header size: 100 B
 Max header delay: 500 ms
 GC depth: 50 rounds
 Sync retry delay: 10,000 ms
 Sync retry nodes: 10 nodes
 batch size: 520 B
 Max batch delay: 10,000 ms

 + RESULTS:
 End-to-end TPS: 3,817 tx/s
 End-to-end BPS: 1,954,120 B/s
 End-to-end latency: 153 ms
-----------------------------------------
```

Listing 6.11: HotStuff Bechmarking Results

```
-----------------------------------------
 SUMMARY:
-----------------------------------------
 + CONFIG:
 Faults: 0 nodes
 Committee size: 10 nodes
 Input rate: 4,000 tx/s
 Transaction size: 512 B
 Execution time: 50 s


 Consensus max payloads size: 32 B
 Consensus min block delay: 0 ms
 Mempool max payloads size: 500,000 B
 Mempool min block delay: 0 ms


 + RESULTS:
 End-to-end TPS: 3,997 tx/s
 End-to-end BPS: 2,046,566 B/s
 End-to-end latency: 85 ms
-----------------------------------------
```

To perform our benchmarking, ten servers (`mpc-0, ..., mpc-9`) for running replicas and one administrative server (`checker`) for handling the benchmarks are utilized within a LAN and are mutual reachable through direct communication. Narwhal or HotStuff instantiations are performed only within the docker container to isolate it from the rest of the host system.

As shown in Figure 6.1, the `checker` is the handler of the experiment, which is responsible for generating the configuration files, transferring the configuration files to `mpc-x`, and starting all `mpc-x` containers for benchmarking. Experiments are controlled by configuration files, and each file is tailored to a specific scenario. Under Scenario 2, for instance, checker creates a `fulty.json` file that is distributed to all servers to tell them which server will terminate its replicas at which moment.

Every server has its replica running in a container, and that replica will read its configuration file to determine its behaviour. The parameter `nodes` determines how many replicas there will be in the container on each server, and the client processes will submit

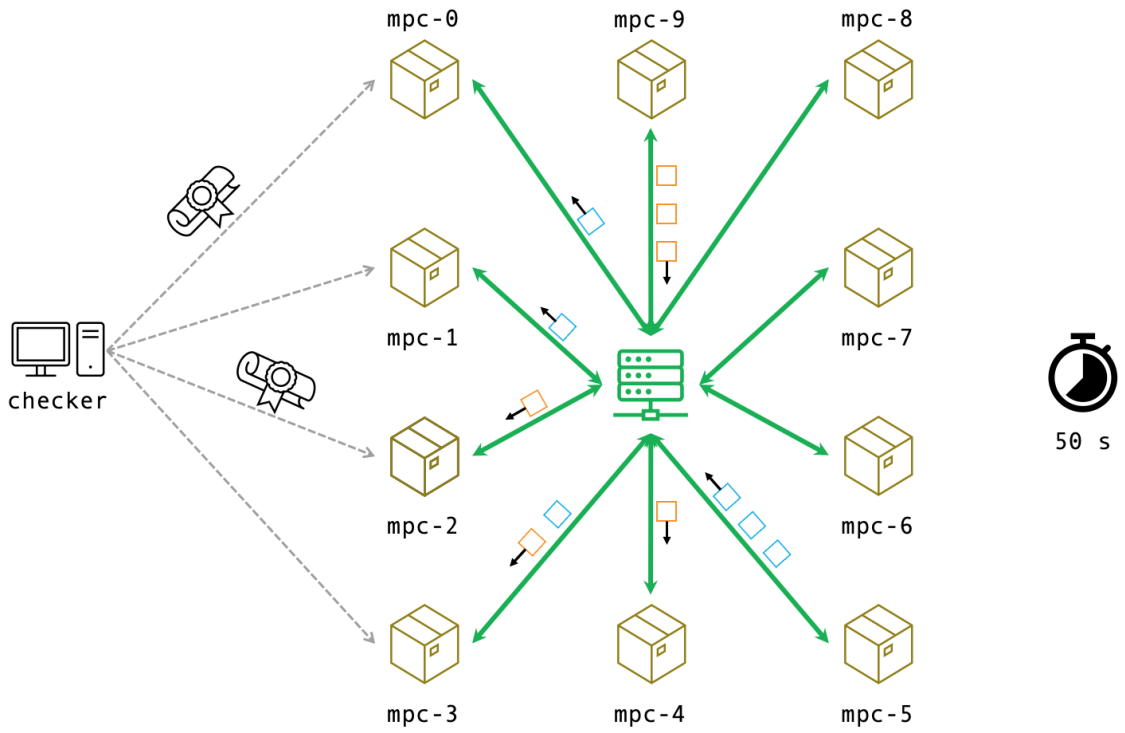


Figure 6.1.: The workflow of benchmarking: `checker` manages 10 Docker containers to run atomic broadcast processes that communicate directly with each other to process transactions, reach consensus, and implement an SMR

transactions to the replicas in the same container at the rate specified by the parameter `rate`; the replicas that receive the transaction requests will exchange messages (orange and blue boxes in the figure 6.1) and reach consensus with the communication partners specified in the `committee.json` file.

During the consensus phase, every replicas takes the necessary action, such as signing or broadcasting, based on the type and content of a message it received. In addition, the replica keeps a local log file that details the steps it took to process each transaction, including the time it was received, the number of rounds it went through, and the time it was committed.

At the end of the experiment, the log files from each service are transferred to the checker server, where they are assembled to analyse the overall performance of the experiment and generate a short summary as shown in the lists 6.10 and 6.11 above. In addition, some important information in the summary report is permanently stored in a database file `results.db` for later analysis and graphical presentation of performance. This includes entries like TPS, Latency, Committee size, and so on.

For the same configuration, we perform 20 rounds of experiment and save the results of each experiment in the database. For stability and efficiency reasons, we decided that the duration of each experiment is 50 seconds. The results of the 20 rounds for each configuration are given a 95% confidence interval, e.g., the 95% confidence interval for the average TPS.

In the next section, we show the details of the three scenarios and describe how the experiments are performed in that scenario: The best case, fail-silent and communication delay.

6.2. Scenarios Design

In this thesis, we set up three scenarios for experiments to observe how the external environment affects the performance of Narwhal and HotStuff. The first one is a happy scenario, which is call the best case. We assume that all parts of the system are working perfectly and without any mistakes. This scenario is quite important as the baseline to show the most normal behaviours of both protocols. We may examine how the two protocols actually function and what performance they can achieve under typical operating conditions by testing Scenario 1. In Scenario 2 we will assume there are parts of the replicas that will be faulty, which is quite possible to happen in the real-world, and then observe how the performance will change accordingly. This allows us to understand the robustness of the system in the face of the risk of faults. We simulate a network instability environment inside Scenario 3, especially with increased communication delay, to try to understand how the two algorithms handle this problem.

These three experimental scenarios are important because they can be used to test the performance and robustness of two atomic broadcasts, which are the critical metrics in the design of distributed fault-tolerant systems. Each of these three scenarios is explained in detail below.

6.2.1. Scenario 1: The Best Case

As the basic benchmarking, we assume that Scenario 1 is a happy case where all the replicas work perfectly and make progress in performing their duties for committing the client's transactions, as showed in figure 6.2. Each replica has its own client, which is responsible to submit transactions at the speed `rate/nodes`. A replicas and its client are inside the same container, and are able to exchange massages without the need for any other external device. A replica is connected with other replicas through peer-to-peer network and communicate directly.

To balance the workload across the 10 available servers, we instantiate the same number of replicas on each server. Each replica receives the input transactions at the shared rate from its client on the same server. Therefore, if 10 replicas are needed to run the protocol, then one replica and one client reside in the docker container on each server; if 20 replicas are needed, then two replicas and two clients reside on each server.

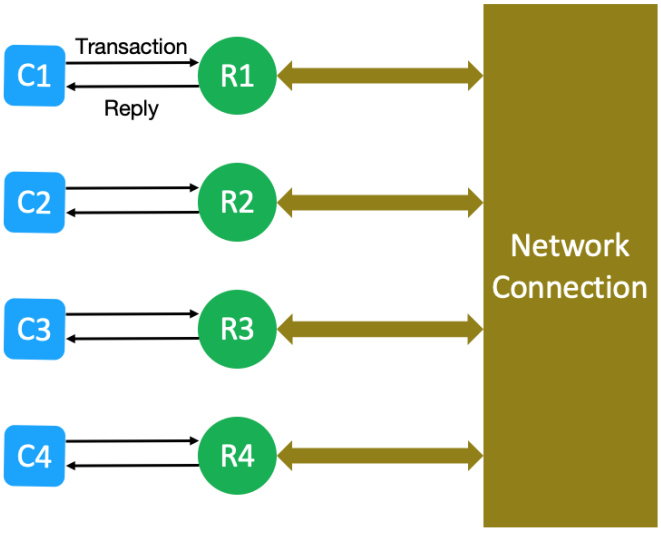


Figure 6.2.: Scenario 1: The best case. Each client sends the transaction to its replica, and waits for the replies from them; all replicas communicates with each other through peer-to-peer network connection and collaborates to reach consensus over how processing the submitted transactions from clients.

The implementation of Narwhal has a restriction to avoid to low input rate, which is found as 40 tx/s per client after testing. To support at most 100 replicas for benchmarking, we set the low bound of input rate is 4000 tx/s. Since Narwhal and HotStuff are both set up to handle only one transaction per consensus round, we decided on a high bound of 10,000 tx/s for input rate to ensure the system is not overloaded too much. Thus, the Scenario 1 has 70 configurations with `nodes = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]` and `rates = [4000, 5000, 6000, 7000, 8000, 9000, 10000]`.

For a configuration with, for example, `node = 40` and `rate = 6000` tx/s, we repeat the benchmarking for 20 rounds and set the `duration = 50` seconds for each round. To complete the whole experiment for Scenario 1 of both protocols, at least $2 \cdot 7 \cdot 10 \cdot 20 \cdot 50$ s $\approx$ 40 hours.

By designing Scenario 1 and specifying the configurations, we investigate how high the TPS both protocols can achieve with a fixed number of replicas and increasing the input rate, how well they maintain the scalability when increasing the number of replicas with a fixed input rate.

6.2.2. Scenario 2: Fail-Silent Benchmarking

In Scenario 2, we look into how faulty replicas can take effect on the performance of Narwhal and HotStuff. Figure 6.3 depicts this scenario, and the red-marked replica is the faulty one in the system; it no longer responds to any messages and can no longer send any messages out. However, the other replicas can't detect whether the red replica is crashed down or just taking a long time to process messages inside a certain bound of time.

After examining the results (section **??**) of Scenario 1, we observed that Narwhal and HotStuff maintained the most stable performance for all replicas from 10 to 100 at input `rate = 10000` tx/s. Therefore, we decide to set 10000 tx/s as the input rate for Scenario 2 and set `nodes = [10, 30, 50, 70, 100]`. We specify two distinguish cases for scenario 2, they are

- **Fixed** number of faulty replicas: Before the start of benchmarking, checker generates a faulty.json file to specify a fixed number of replicas which are designated randomly as faulty. At the **5th** second after benchmarking, these faulty replicas are killed in the background to simulate the fail-silent behaviour. Given the total number of replicas, we try all the possible numbers of faulty replicas. With `node = 3f+1`, there are `f+1` options of the fixed number: `[0, 1, ..., f]`. For example, `nodes = 50` with `f = 16`, then we set `17` possible values for the number of faulty replicas. Every feasible benchmarking is performed by iteratively increasing this number up to f. The performance metrics are recorded to determine how this number affects the behaviours of Narwhal and HotStuff, the benchmarking for each option repeats 20 rounds to reduce the statistical variance.
- **Growing** number of faulty replicas: Before the start of benchmarking, f replicas are designated randomly as faulty. Each of them is killed at a randomly specified time point between at least **5th** second after the start of benchmarking to at most **(duration - 10)th** second before the end of benchmarking. Assumed `node = 10` with `f = 3`, it is possible that replicas `r_3` crashed at 6th second, `r_8` at 35th second and `r_1` at 42th second.

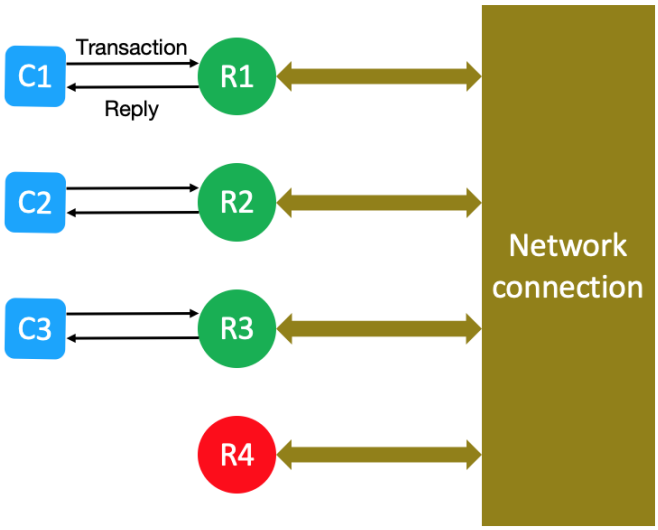


Figure 6.3.: Scenario 2: The Fail-silent case. Parts of the replicas will be faulty during benchmarking, they cannot participate in the consensus nor response to the clients.

Narwhal deploys a zero-message flip coin mechanism to randomly pick leaders retrospectively for reaching consensus, while the leaders in HotStuff are deterministic selected,

such as $l = viewNumber \mod n$. To that end, we're curious to see how they perform when confronted with faulty replicas or even faulty leaders. The time needed to complete Scenario 2 is $2 \cdot ((4+1)+(10+1)+(17+1)+(24+1)+(34+1)) \cdot 20 \cdot 50$ s $\approx$ 52 hours.

6.2.3. Scenario 3: Communication Delay

Another possible situation in the real-world is unstable network connection. The communication delay between two ends could be increased significantly due to unknown reasons, and it is unpredictable. In Scenario 3, we defer message delivery to increase the communication delay by using `tc netem`, which means all messages sent from the container to the network interface (`eth0`) on the host machine will be delayed on the network card for a certain period. This delay effect begins at the exact same instant as the benchmarking begins.

The figure 6.4 shows this situation that the message passing between any two replicas will be deferred (red clock symbol) for a certain period of time. But the communication between the replica and its client inside the same container is not affected, any transaction submit by client will be sent immediately to its responding replicas.

We tried the delay range from zero ms exponentially to 8192 ms, i.e., `delays = [0, 2, 4, 8, 16, 32, 64, 128, ..., 4096, 8192]`, for `nodes = [10, 30, 50, 70, 100]`. To prevent the system from being overloaded by cumulative submitted transactions during the communication delay (clients are not affected by the delay), we set input `rate = 4000` tx/s.

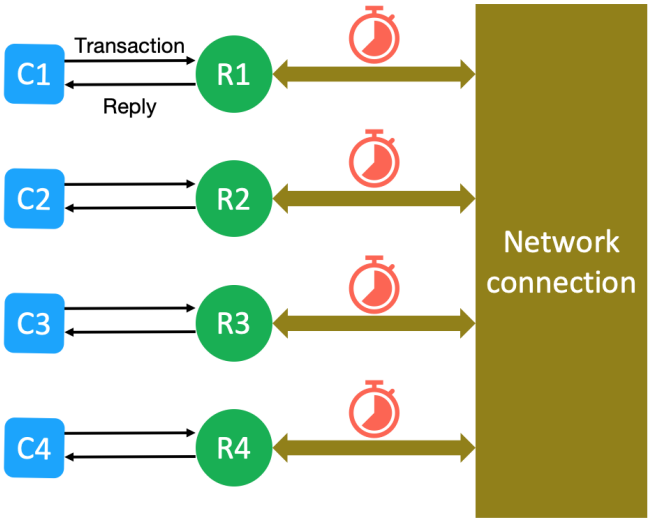


Figure 6.4.: Scenario 3: The Delay case. All the communication between any two replicas will be delayed for a certain time period, to simulate a worse network condition.

We believe that Narwhal can tolerate a greater communication delay than HotStuff does because of the different timing assumptions made by the two systems. Therefore, the purpose of the Scenario 3 is to verify our assumptions. To complete this scenario, we need at least $2 \cdot 5 \cdot 14 \cdot 2050$ s $\approx$ 40 hours.

7. Evaluation

To show how Narwhal and HotStuff respond differently to the experimental environment, we exhibit performance illustrations after parsing the log files for all three scenarios in this chapter. First, we calculate what performance of the two atomic broadcasts is achieved in the scenario 1 with different combinations of number of replicas and input rates. We also graphically present the variation of TPS and Latency with respect to the number of replicas and the input rate, respectively. Then, we try to analyse and explain the possible reasons behind the variation in performance of the two atomic broadcasts. Once we have analysed the experimental results from Scenario 1, we will use that information to determine the optimal parameters setting for Scenario 2 and Scenario 3, such as the fixed number of replicates and input rate. In Scenario 2 we are more concerned with the number of faulty replicas, and in Scenario 3 we are mainly concerned with the length of the communication delay.

7.1. Analysing the Results of Scenario 1

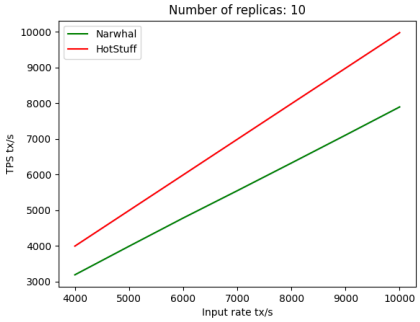
As mentioned before, we investigate Scenario 1 with different number of replicas, all is correct, and different input rate, to quantify the performance of Narwhal and HotStuff by measuring their TPS and latency. Both Narwhal and HotStuff are set up to handle a single transaction per consensus round by restricting their payload size (lists 6.3 and 6.5) to avoid the optimization of batch processing approaches.

As the baseline, the figures 7.1 illustrate the throughput and latency of Narwhal and HotStuff for increasing input rate from 4000 tx/s to 10000 tx/s when the number of replicas is fixed as 10. When we increase the input rate from 4000 tx/s to 10000 tx/s for Narwhal and HotStuff, the throughput (TPS) can always scale linearly with input rate. HotStuff can almost achieve the TPS approximately to the input rate, while Narwhal only reaches the $80\% - 85\%$ of the input rate.

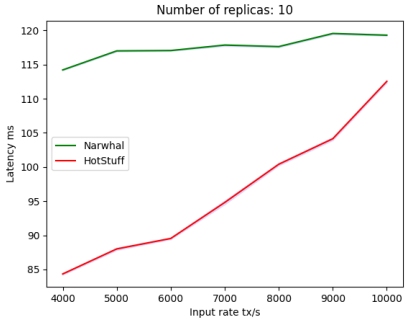
The performance gap between the input rate and the actual TPS that is capable of achieving by Narwhal is caused partly by its retrospective consensus: A block into the DAG can only be considered as committed at least after one wave (three rounds) has been completed. Right at the end of the 50s-duration, the consensus mechanism of coin flips cannot be applied to reach consensus over the transactions that are sealed into blocks in the last few rounds, because that wave is not yet completed.

The immediate finality of HotStuff, on the other hand, guarantees that every block on the blockchain is committed, hence it never runs into this problem (or this huge gap).

By conducting additional experiments to verify our hypothesis, we tried durations from 50s to 200s in Scenario 1, with input rates ranging from 250 tx/s to 5000 tx/s (step is 250) and 10 replicas. For each duration and each input rate, we repeat the benchmarking for 10



(a) TPS with 10 replicas



(b) Latency with 10 replicas

Figure 7.1.: The baseline benchmarking for Scenario 1

rounds. Then, the gap between the real TPS and the input rate is calculated to measure the effect of duration on the TPS gap.

This figure 7.2 depicts their relationship.

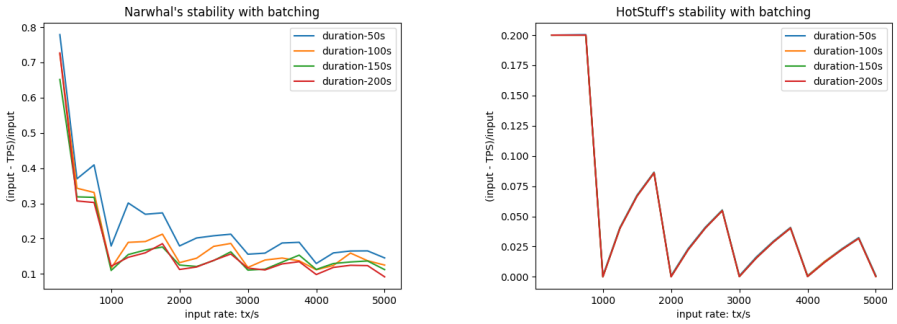


Figure 7.2.: The relation between TPS gap and duration with increasing input rate

From these two images, we can deduce that a reduced TPS gap is achieved by raising the input rate in both Narwhal and HotStuff, and that the TPS gap decreases with increasing duration for Narwhal but has essentially no effect on the TPS gap for HotStuff. Another interesting phenomenon is that both curves show jagged features, with HotStuff being more pronounced. This jagged fluctuation stems from the matching relationship between the size of the payload and the input rate, where the payload size is most efficient for reaching consensus when the input rate is a whole thousand (x000). This payload size is initially configured as `batch_size = 500000` for Narwhal, and `max_payload_size = 500000` for HotStuff. Therefore, we see an increase in gap around the whole thousand.

What attracts our interest more is how Narwhal and HotStuff were able to engage the consensus in the face of more replicas. Therefore, we keep increasing the number of replicas from 10 up to 100. To balance the payload, each server is assigned with the same number of replicas. That is from 1 to 10.

We don't present here all possible combinations between number of replicas and input rates, only partial results are demonstrated in figure 7.3 to exhibit the effect that increasing

the replica number can have on the throughput. The complete results are attached on Appendix A.

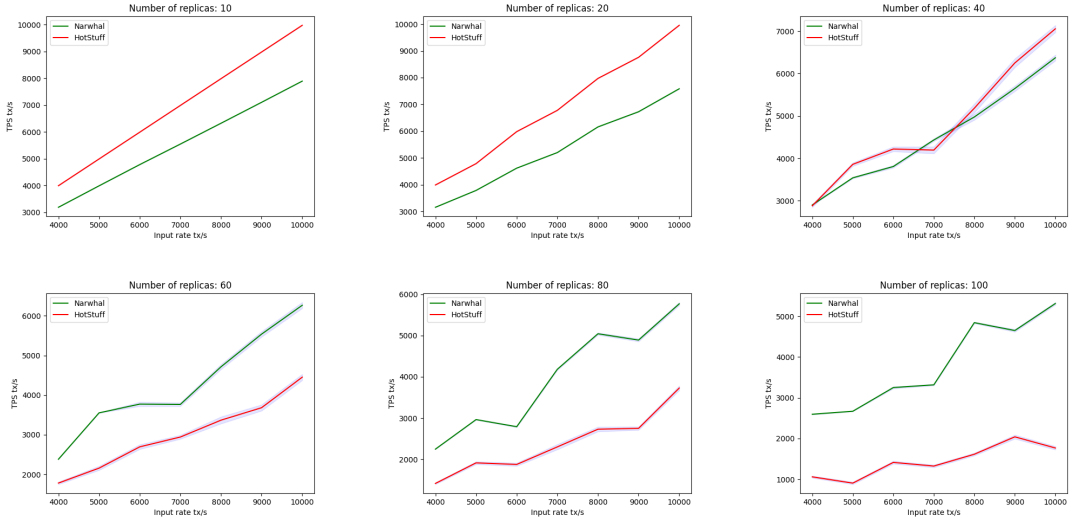


Figure 7.3.: TPS against input rate with fixed replicas for Scenario 1

Compared to 10 replicas, it is reasonable that both Narwhal and HotStuff suffer from TPS reduction when engaging more replicas: The more replicas, the higher message complexity to reach consensus. With more than 40 replicas, HotStuff's TPS degrades substantially more than Narwhal's, despite being tuned to establish a linear relation between communication complexity and the number of replicas. We believe this is because asynchronous atomic broadcasts can accommodate more message exchanges than partially synchronous atomic broadcast. And DAG is naturally suited for parallel processing of transactions, while blockchain is only capable of linear processing.

It is observable that the more replicas, the greater Narwhal's advantage. Even with 100 replicas, Narwhal is still able to achieve TPS of more than half of its input rate, while HotStuff can only reach a maximum of 2000 tx/s, no matter how high the input rate is.

Thanks to Narwhal's asynchronous design, each replica can make progresses as long as it received enough parent blocks, balancing out the waiting time among all replicas. HotStuff, on the other hand, only leader can propose blocks after it gather enough `NEW-VIEW` messages, which is the bottleneck of the message exchanging.

Due to its retrospective consensus mechanism, Narwhal reaches consensus always with longer latency than HotSutff for 10 replicas. But once we increase the number of replicas, HotStuff's latency grows rapidly and exceeds Narwhal's (except for 100 replicas).

The same thing that happened to Narwhal also happened to HotStuff when we examine the correlation between TPS and replica count as showed in figure 7.4: With fixed input rate, the TPS decreases with growing replicas. HotStuff's TPS experiences much worse degradation than Narwhal, such that Narwhal'TPS is even higher for over 40 replicas, no matter the input rate. The figure 7.4 and 7.5 illustrate that Narwhal's asynchronous algorithm and the parallel structure of the DAG provide better scalability, and Narwhal achieves better TPS and latency than HotStuff when accommodating more replicas to participate in consensus

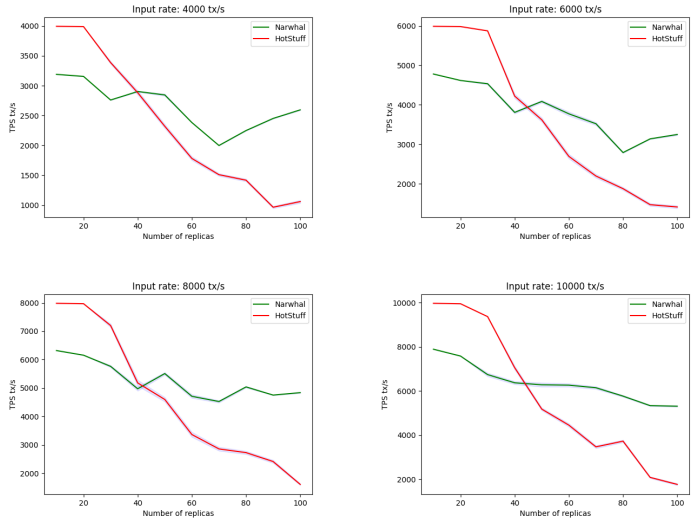


Figure 7.4.: The TPS against replicas with fixed input rate for Scenario 1

Plots of latency against input rate for more than 50 replicas in figure 7.5 exhibit two opposing trends: For Narwhal, latency rises with higher input rate, and for HotStuff, it falls. The reason behind this discrepancy is related to the capability of consensus: Narwhal achieved higher TPS than HotStuff for more than 50 replicas. That is, Narwhal processes more transactions within the benchmarking, resulting in growing latency.

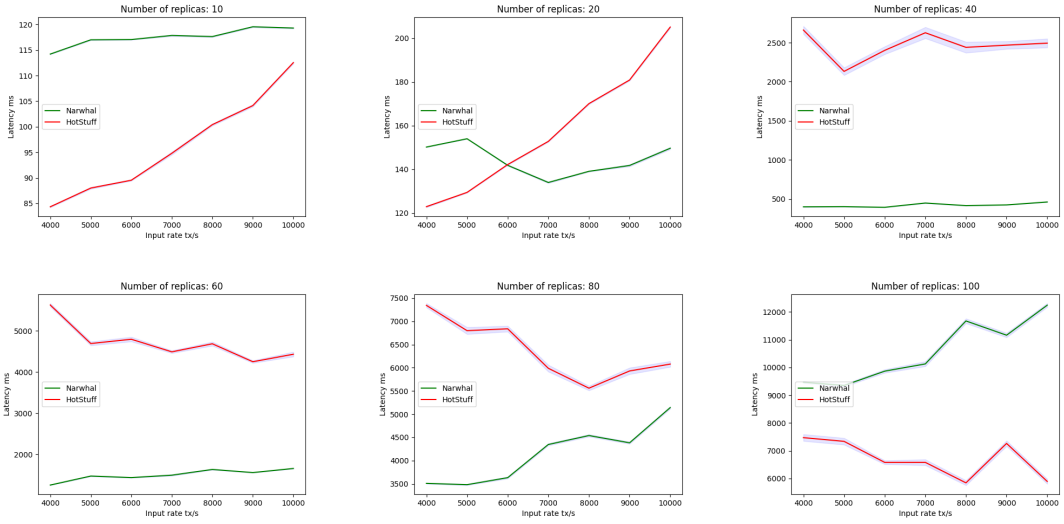


Figure 7.5.: The Latency against input rate with fixed replicas for Scenario 1

We can also see that Narwhal maintains a much more stable latency than HotStuff with the increasing input rate in figure 7.5. Different conclusions emerge, however, when we examine what effect the number of replicas can have on the latency, showed in figure 7.6

If we fix the input rate and keep increasing the number of replicas, we can find that the latency of Narwhal grows exponentially with replicas amount, while HotStuff maintains

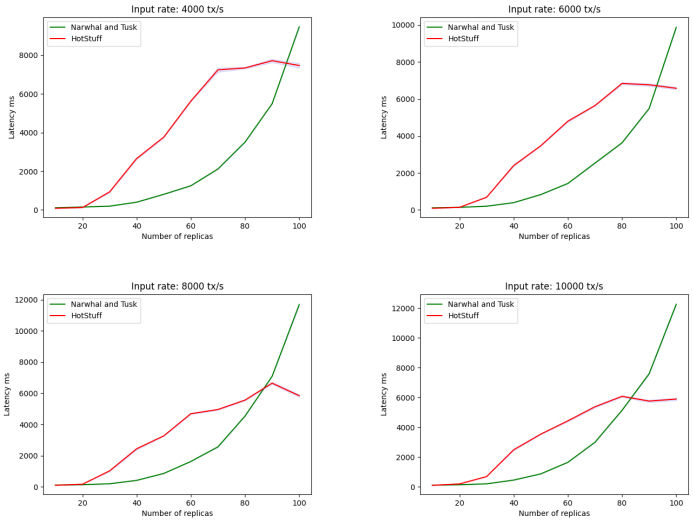


Figure 7.6.: The Latency against replicas with fixed input rate for Scenario 1

a nearly linear growth for latency. This is an effect of the theoretical complexity of communication: Narwhal deploys reliable broadcast for message passing, which has $O(n^2)$ (exponential) complexity, which HotStuff implements a leader with combination of multicast and unicast to realize $O(n)$ (linear) complexity.

7.2. Analysing the Results of Scenario 2

Scenario 2 is not a happy case, but a more practical case for the fault-tolerance system: Some components might be failed silently. At some point after the beginning of the benchmarking, a number of randomly selected replicas will be terminated to imitate this real scenario.

After finishing experiments for Scenario 1, we don't need to try all combinations of input rate and replicas count for Scenario 2. Observing the plots from the section 7.1, we've found that, with 10000 tx/s, both Narwhal and HotStuff can maintain most stable TPS for all possible replicas from 10 to 100. Hence, we choose `rate = 10000` tx/s as the input rate for Scenario 2, and `replicas = [10, 30, 50, 70, 100]`, reducing the experiment complexity. For each possible number of replicas, we distinguished two subcase as described in Chapter 6: the fixed number of faulty replicas, and the growing number of faulty replicas.

In the figure 7.7, the green star stands for the case of growing number of Narwhal faulty replicas, while the red cross stand for HotStuff. It is quite obvious that HotStuff always suffers the TPS's degradation when we increase the fixed number of faulty replicas. Narwhal can keep its overall good TPS when facing the increasing fixed number of faulty replicas until a fast drop happened.

As of this now, we don't know why Narwhal's TPS decreases so precipitously after the fixed number of faulty replicas surpasses an unknown threshold. This Cliff-like TPS degradation occurring solely on Narwhal is still an interesting phenomenon we want to

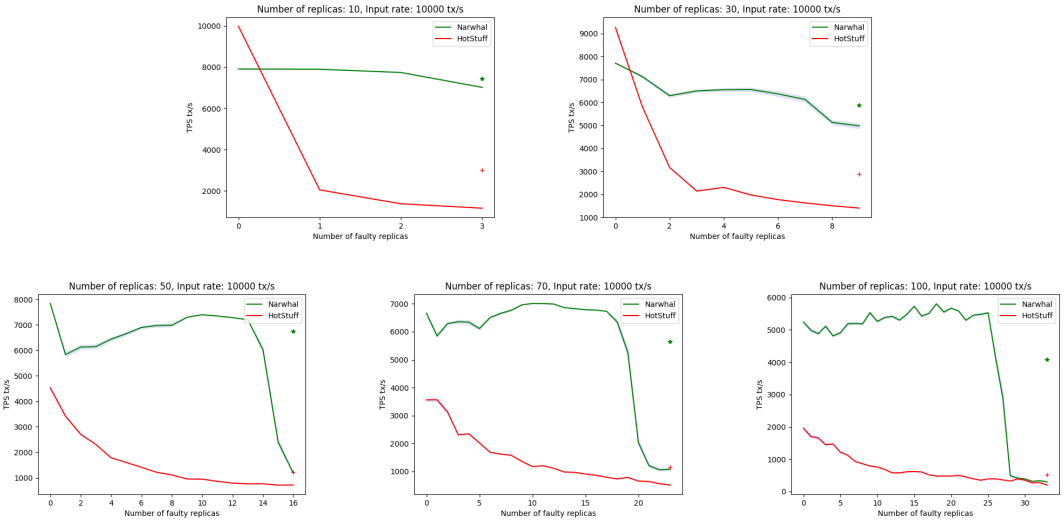


Figure 7.7.: The TPS against faulty replicas for Scenario 2. The green star and red cross stand for the growing faulty replicas for Narwhal and HotStuff respectively, two lines are for the fixed faulty replicas.

figure out. One possible reason is that Narwhal stops consensus and loses liveness after the number of false replica increases to a threshold (less than f). And the very low TPS shown in this figure 7.7 is actually the result of the work done within five seconds after the benchmarking started.

For the case of growing faulty replicas, we notice that both Narwhal and HotStuff can achieve better TPS than the case of fixed f faulty replicas. Because for fixed faulty replicas, all the faulty replicas are terminated to work at the 5-th second after the start of benchmarking. When f is the fixed number, there are only $n - f$ replicas to make progresses between interval `[5, duration]`. For the growing faulty replicas, they are killed one by one at different time point between the interval `[5, duration-10]`. Hence, for the growing case, the average working time of replicas is higher than the fixed case. The effect periods of the faulty replicates varying in length results in different level of TPS.

The plots of latency against faulty replicas in figure 7.8 showed that HotStuff is much more sensitive to the faulty replicas. Because HotStuff has a view-based consensus mechanism to guarantee the liveness property, once a leader on the current view is selected as faulty and killed, then all other correct replicas have to wait and make no progress until a timeout to trigger the `VIEW-CHANGE` process, which results in much longer latency and much worse TPS than Narwhal.

Narwhal, however, doesn't depend on such timeout mechanism for leader election, and any correct replica can keep processing transactions. Once a correct leader is elected retrospectively, the finished work will then be committed as the consensus results. If a randomly selected replica is faulty and doesn't create any block more, then that wave will be committed following a correct leader in the previous wave. Narwhal can suffer no stagnation as long as $2f + 1$ replicas are correct, while HotStuff must stop working for a while once a leader is killed.

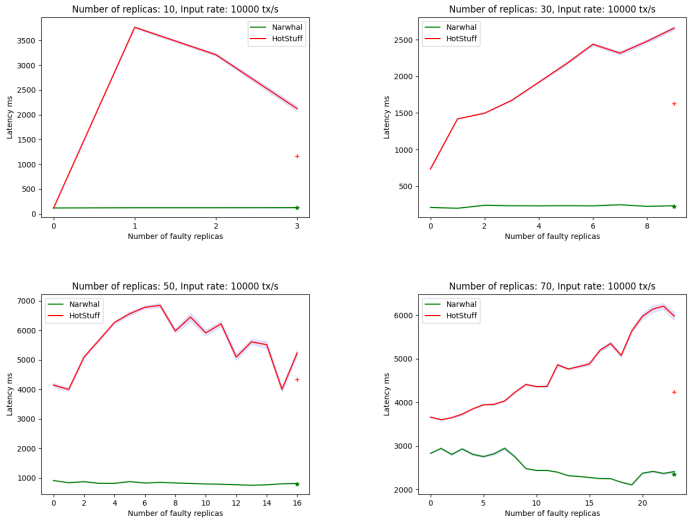


Figure 7.8.: The latency against faulty replicas for Scenario 2. The green star and red cross stand for the growing faulty replicas for Narwhal and HotStuff respectively, two lines are for the fixed faulty replicas.

7.3. Analysing the Results of Scenario 3

When we finished the experiments for Scenario 1 and 2, we get quite a lot of information about their behaviours. To investigate how communication delay will affect their performance, Scenario 3 is designed to simulate an unstable network. In this scenario, we keep increasing the message delay for both atomic broadcast, and observe how they behave differently.

The `timeout` parameter for HotStuff is 1000 ms, while Narwhal has no such setting. Then, we decide the delay range from zero ms exponentially to 8192 ms. That is, `delays = [0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]`.

We observe a similar phenomenon for both Narwhal and HotStuff: Both protocols have slight fluctuations in TPS until the communication delay exceeds a certain threshold; once the communication delay reaches the threshold, TPS drops rapidly to near zero. For HotStuff with 10 replicas, this threshold is exactly as 1024 ms which is longer than the timeout parameter, while Narwhal has a little greater threshold than HotStuff, it is approximately 2048 ms. This threshold shifts to the left on the x-axis as we increase the number of replicas, which is caused by more message exchanges, making the system more sensitive to message delay.

As we mentioned in Section 6.2.3, we reduced the input rate from 10000 tx/s to 4000 tx/s in order to prevent overloading the system. We actually performed the same experiment with the input rate of 10000 tx/s, which resulted in an overall shift of all curves to the left by a few units on the basis of figure 7.9.

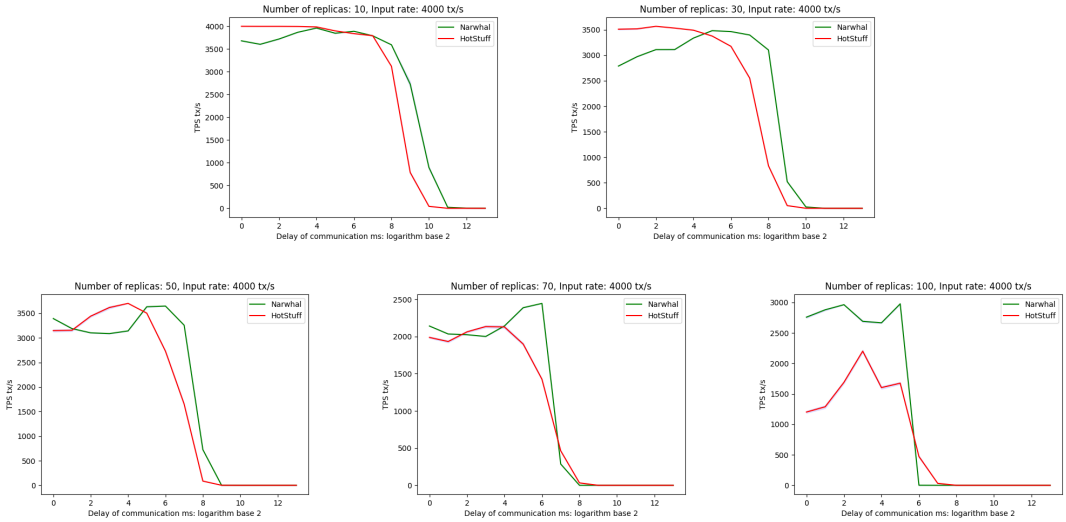


Figure 7.9.: The TPS against communication delay for Scenario 3. The TPS of both Narwhal and HotStuff show a rapid decline after the communication delay exceeds a certain limit, when the system is stuck and cannot continue to reach consensus.

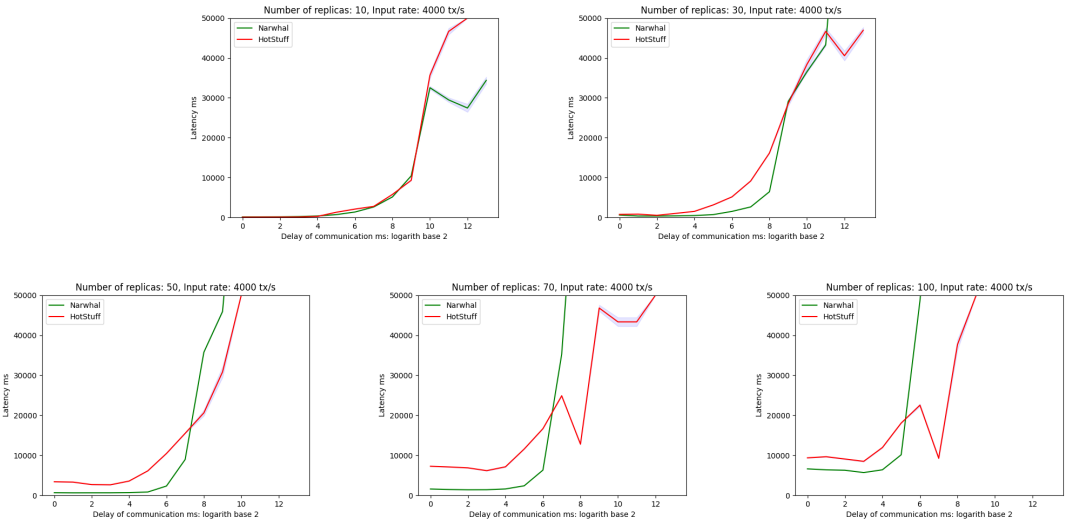


Figure 7.10.: The latency against communication delay for Scenario 3. The latency curves of Narwhal and HotStuff look exponentially increasing, but this visual effect comes from the fact that the x-axis coordinates are logarithmic.

As showed in the latency plots of figure 7.10, both atomic broadcasts experience exponentially growing consensus latency once we keep increasing the communication delay. Because the communication delay is growing exponentially, it is quite normal for the consensus latency to grow exponentially along with it. Once the delay grows to the point where both atomic broadcasts are stopped working and lost their liveness, their consensus latency should theoretically be infinite. However, we cannot show an infinite value in the graph, so we decided to represent that infinity by twice the duration: 100s. Thus, we can

find that the rightmost part of the latency curves of Narwhal and HotStuff in figure 7.10 is not shown in the plots, because their real number is 100s, while the upper limit of the y-axis is only 50s

In fact, an approximately linear rather than exponential relationship exists between latency and the communication delay. To better illustrate the nuances of the latency variation, we utilize the logarithm of 2 as the x-axis, resulting in plots that increase exponentially.

7.4. Summary and Discussion

Several hundred hours of experiments have been conducted, and this chapter have presented and analysed some of the most noteworthy findings. Our experiments in Scenario 1 exposed the scalability issue with atomic broadcast, i.e., the performance of both synchronous and asynchronous algorithms is invariably negatively affected as the number of replicas participating in consensus increases, with TPS reducing and latency getting longer. From the most classical PBFT, to today's linear complexity HotStuff and asynchronous Narwhal, all suffer from this scalability challenge. According to some recent papers [Fu+22], and [Liu+18], perhaps Trusted Execution Environments (TEEs) can help implement atomic broadcast techniques with better scalability.

Furthermore, through the experiments in Scenario 2, we find a significant difference in the performance impact of faulty replicas on asynchronous and partially synchronous atomic broadcasts. The throughput degradation of HotStuff is significantly greater than Narwhal in the face of an increasing number of faulty replicas. This is because the unresponsiveness of the faulty replica frequently triggers the interrupter to change view, incurring significant additional communication overhead. Narwhal maintains decent throughput thanks to its purely asynchronous communication and the ability to process transactions in parallel due to DAG data. Because of the difference in leader election and number of messages exchanged, the latency of Narwhal and HotStuff changes in the exact opposite way when facing the faulty replicas. Narwhal reaches consensus faster because of the reduced number of messages exchanged, while HotStuff has a longer latency due to the extra waiting time caused by the view change. We can say that partially synchronous atomic broadcasts are more sensitive to faulty replicas than purely asynchronous atomic broadcasts. Lastly, the results of the Scenario 3 experiments show that neither partially synchronous nor purely asynchronous atomic broadcasts can handle message delays of arbitrary long. HotStuff keeps triggering a timeout error, leading to complete no transactions, and Narwhal can not tolerant message delay once it exceeds some threshold, although it has no such timeout mechanism.

Our experimental setup does not simulate more realistic scenarios, e.g., the message delay is different for each server, and the effect period of the delay can be set as a dynamic interval instead of the duration of the benchmarking. Another real-world scenario is network partitioning, where different components of the system are separated in different subnets, and the components in different subnets can no longer communicate with each other. It is still an open question whether atomic broadcast can still guarantee its safety and liveness. In addition, packet loss during transmission is another common network problem,

which is also worth being carefully explored to discover what impact on the performance of the atomic broadcast can be. All these efforts should continue to be completed in the future work.

In our experiments, we adopt a special configuration that allows both Narwhal and HotStuff to process only one transaction per consensus round, removing the optimization effect of different block sizes and batching processing. This is actually a normalization process that facilitates comparison of how different atomic broadcasts really perform relative to each other. This method is also suitable for the benchmarking of other atomic broadcast technologies. The better performance of asynchronous atomic broadcast verified by experimental results benefits from the parallelism of the underlying DAG structure, which is suitable for the parallel growth of data and can be applied to the design of a wider range of distributed algorithms.

8. Conclusion

We explained the fundamental concepts in the field of fault-tolerant system, such as Byzantine Generals Problem, the State Machine Replication, fault models and timing assumptions. Then, we introduced the technique Atomic Broadcast, which can be utilized to implement the State Machine Replication, with its formalized properties. In this thesis, we choose to focus on two typical types of atomic broadcasts: Narwhal, which operates in a fully asynchronous context, and HotStuff, which operates in a partially synchronous setting.

By analysing their papers, we not only reveal the design principles of the Narwhal mempool protocol and the way it collaborates with Tusk, and demonstrate its good parallelism; we also explain the three-phase communication flow, which implementing linear message complexity, and the triggering mechanism of view change. After understanding the theoretical principles of these two atomic broadcasts, we developed an experimental framework that was deployed on ten servers to benchmark the performance of the open-sourced implementations of these two algorithms in three specified scenarios.

The practical experiments conducted in three scenarios have demonstrated that Narwhal achieves much better scalability and stability than HotStuff, it can accommodate more replicas in the consensus process and maintain a decent TPS and latency. Despite the claim in the paper that HotStuff implements linear message complexity, its performance is still strictly limited by the number of replicas, and it cannot admit too many replicas to participate in consensus together without severe performance degradation. Especially, as the number of faulty replicas grows, Narwhal's overall performance remains good (much better than HotStuff's), allowing it to be employed for real-world at massive scale, while HotStuff suffers from great performance degradation.

Therefore, asynchronous systems like Narwhal are becoming more realistic and useful in the real world, even when parts of individual replicas are problematic. Applying Narwhal to the real world is a more secure and more pragmatic option than HotStuff. In other words, asynchronous atomic broadcasts are more likely to be applied to real systems in the future than partially synchronous atomic broadcasts due to their good scalability and robustness.

In addition, both Narwhal and HotStuff suffer from performance degradation when communication delay keeps increasing, although Narwhal can withstand slightly longer delays than HotStuff. This shows that Narwhal is not a fully asynchronous atomic broadcast that can withstand arbitrary delay, as was claimed in the publication, and that coping with a communication delay of arbitrary length is still a challenging task.

Bibliography

[Abd+05] Michael Abd-El-Malek et al. "Fault-Scalable Byzantine Fault-Tolerant Services". In: *SIGOPS Oper. Syst. Rev.* 39.5 (Oct. 2005), pp. 59–74. ISSN: 0163-5980. DOI: 10.1145/1095809.1095817. URL: https://doi.org/10.1145/1095809.1095817.

[Avi+04] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Hoboken, NJ, USA: John Wiley and Sons, Inc., 2004. ISBN: 0471453242.

[Bai16] Leemon Baird. "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance". In: *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* 34 (2016).

[Bau+19] Mathieu Baudet et al. "State machine replication in the libra blockchain". In: *The Libra Assn., Tech. Rep* (2019).

[BSA14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. *State Machine Replication for the Masses with BFT-SMART*. 2014. DOI: 10.1109/DSN.2014.43.

[BT85] Gabriel Bracha and Sam Toueg. "Asynchronous Consensus and Broadcast Protocols". In: *J. ACM* 32.4 (Oct. 1985), pp. 824–840. ISSN: 0004-5411. DOI: 10.1145/4221.214134. URL: https://doi.org/10.1145/4221.214134.

[CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2011. ISBN: 3642152597.

[CL+99] Miguel Castro, Barbara Liskov, et al. "Practical Byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999, pp. 173–186.

[CL02] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: 20.4 (2002). ISSN: 0734-2071. DOI: 10.1145/571637.571640. URL: https://doi.org/10.1145/571637.571640.

[Cle+12] Allen Clement et al. "On the (limited) power of non-equivocation". In: *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. 2012, pp. 301–308.

[Cra20] Tyler Crain. *Experimental Evaluation of Asynchronous Binary Byzantine Consensus Algorithms with $t < n/3$ and $O(n^2)$ Messages and $O(1)$ Round Expected Termination*. 2020. DOI: 10.48550/ARXIV.2004.09547. URL: https://arxiv.org/abs/2004.09547.

[Cra21] Karl Crary. "Verifying the hashgraph consensus algorithm". In: *arXiv preprint arXiv:2102.01167* (2021).

[Cri+95] Flaviu Cristian et al. "Atomic broadcast: From simple message diffusion to Byzantine agreement". In: *Information and Computation* 118.1 (1995), pp. 158–179.

[Dan+21] George Danezis et al. "Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus". In: *CoRR* abs/2105.11827 (2021). arXiv: 2105.11827. URL: https://arxiv.org/abs/2105.11827.

[DCK15] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. "Resource-efficient Byzantine fault tolerance". In: *IEEE transactions on computers* 65.9 (2015), pp. 2807–2819.

[DGV05] Partha Dutta, Rachid Guerraoui, and Marko Vukolic. *Best-case complexity of asynchronous Byzantine consensus*. Tech. rep. Technical Report EPFL/IC/200499, EPFL, 2005.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey". In: *ACM Comput. Surv.* (Dec. 2004), pp. 372–421.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[Fu+22] Xiang Fu et al. "Teegraph: A Blockchain consensus algorithm based on TEE and DAG for data sharing in IoT". In: *Journal of Systems Architecture* 122 (2022), p. 102344. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102344. URL: https://www.sciencedirect.com/science/article/pii/S1383762121002356.

[Gir+22] Neil Giridharan et al. "Bullshark: DAG BFT Protocols Made Practical". In: *arXiv preprint arXiv:2201.05677* (2022).

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg, 2015.

[Guo+20] Bingyong Guo et al. "Dumbo: Faster asynchronous bft protocols". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 803–818.

[Kei+21a] Idit Keidar et al. *All You Need is DAG*. 2021. DOI: 10.48550/ARXIV.2102.08325. URL: https://arxiv.org/abs/2102.08325.

[Kei+21b] Idit Keidar et al. "All you need is dag". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 2021, pp. 165–175.

[KMS19] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. *Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures*. Cryptology ePrint Archive, Paper 2019/1015. https://eprint.iacr.org/2019/1015. 2019. URL: https://eprint.iacr.org/2019/1015.

[Liu+18] Jian Liu et al. "Scalable byzantine consensus via hardware-assisted secret sharing". In: *IEEE Transactions on Computers* 68.1 (2018), pp. 139–151.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.

[Mil+16] Andrew Miller et al. "The honey badger of BFT protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 31–42.

[MMR14] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. "Signature-free asynchronous Byzantine consensus with t< n/3 and O (n2) messages". In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 2014, pp. 2–9.

[Nak09] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[NET21] Martin Nischwitz, Marko Esche, and Florian Tschorsch. "Bernoulli meets pbft: Modeling bft protocols in the presence of dynamic failures". In: *2021 16th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE. 2021, pp. 291–300.

[PGS98] Fernando Pedone, Rachid Guerraoui, and André Schiper. "Exploiting atomic broadcast in replicated databases". In: *European conference on parallel processing*. Springer. 1998, pp. 513–520.

[PSL80] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (Apr. 1980), pp. 228–234. ISSN: 0004-5411. DOI: 10.1145/322186.322188. URL: https://doi.org/10.1145/322186.322188.

[Sch84] Fred B. Schneider. "Byzantine Generals in Action: Implementing Fail-Stop Processors". In: *ACM Trans. Comput. Syst.* 2.2 (May 1984), pp. 145–154. ISSN: 0734-2071. DOI: 10.1145/190.357399. URL: https://doi.org/10.1145/190.357399.

[Sch90] Fred B. Schneider. "The state machine approach: A tutorial". In: *Fault-Tolerant Distributed Computing*. Ed. by Barbara Simons and Alfred Spector. New York, NY: Springer New York, 1990, pp. 18–41. ISBN: 978-0-387-34812-4.

[Sho00] Victor Shoup. "Practical threshold signatures". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2000, pp. 207–220.

[Sta+19] Chrysoula Stathakopoulou et al. "Mir-bft: High-throughput robust bft for decentralized networks". In: *arXiv preprint arXiv:1906.05552* (2019).

[Ver+11] Giuliana Santos Veronese et al. “Efficient byzantine fault-tolerance”. In: *IEEE Transactions on Computers* 62.1 (2011), pp. 16–30.

[Xu21] Wenbo Xu. “Hybrid Fault Tolerant Consensus in Wireless Embedded Systems”. PhD thesis. Braunschweig, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2021.

[Yin+18] Maofan Yin et al. “HotStuff: BFT consensus in the lens of blockchain”. In: *arXiv preprint arXiv:1803.05069* (2018).

[Zha+21] Jiashuo Zhang et al. “Efficient Byzantine Fault Tolerance using Trusted Execution Environment: Preventing Equivocation is only the Beginning”. In: *arXiv preprint arXiv:2102.01970* (2021).

A. Appendix 1: Complete Results for Scenario 1

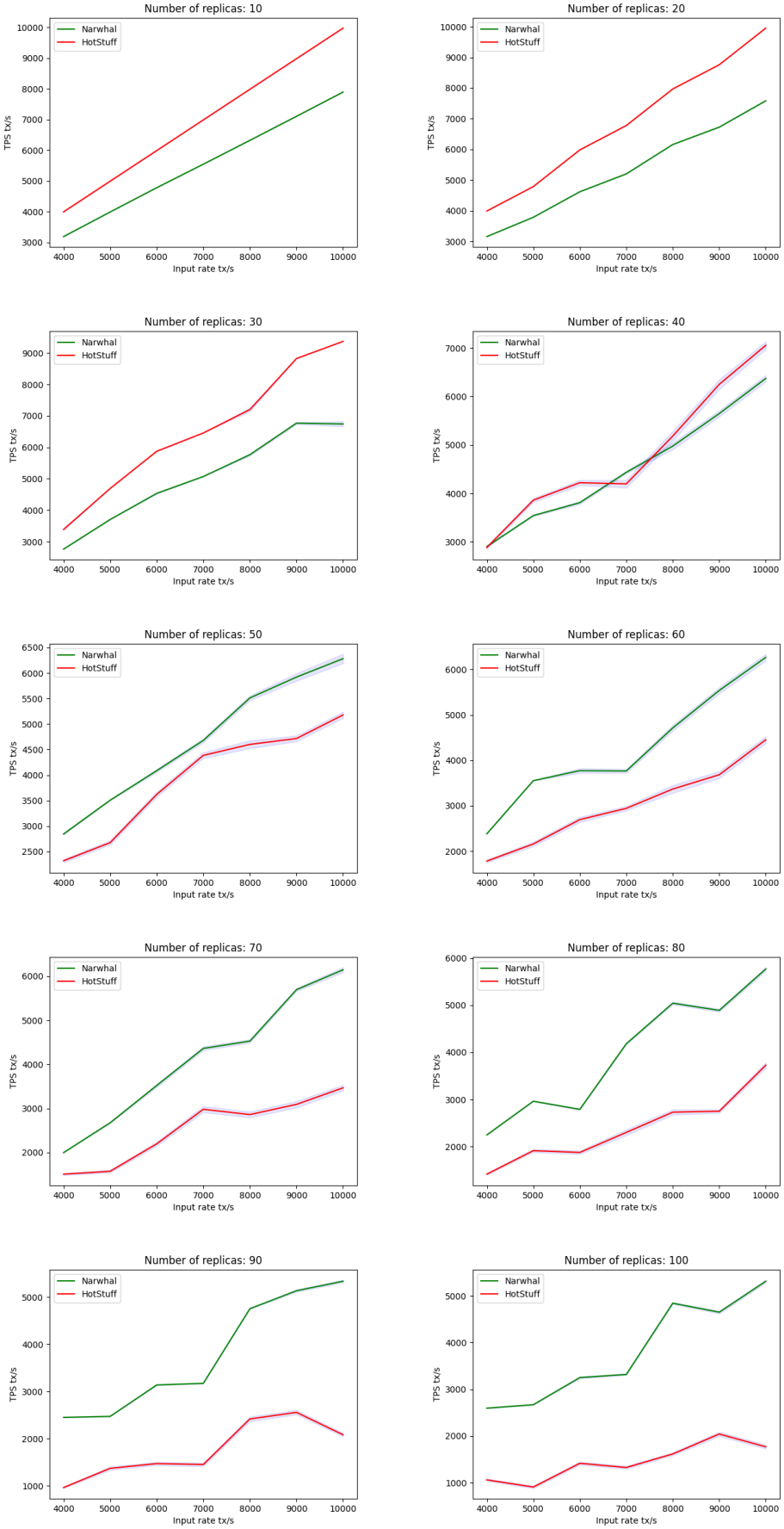


Figure A.1.: The TPS for Scenario 1

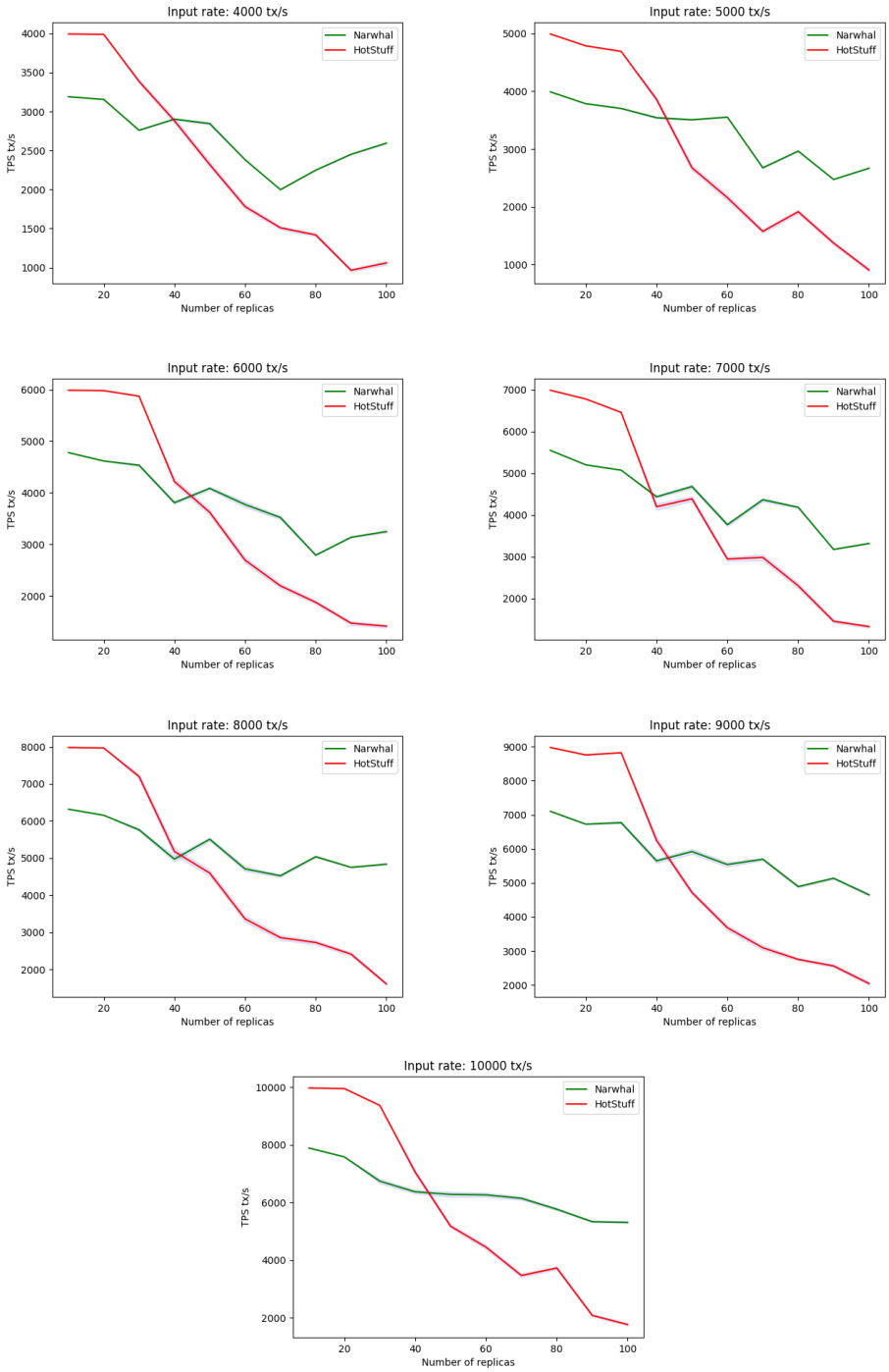


Figure A.2.: The latency for Scenario 1

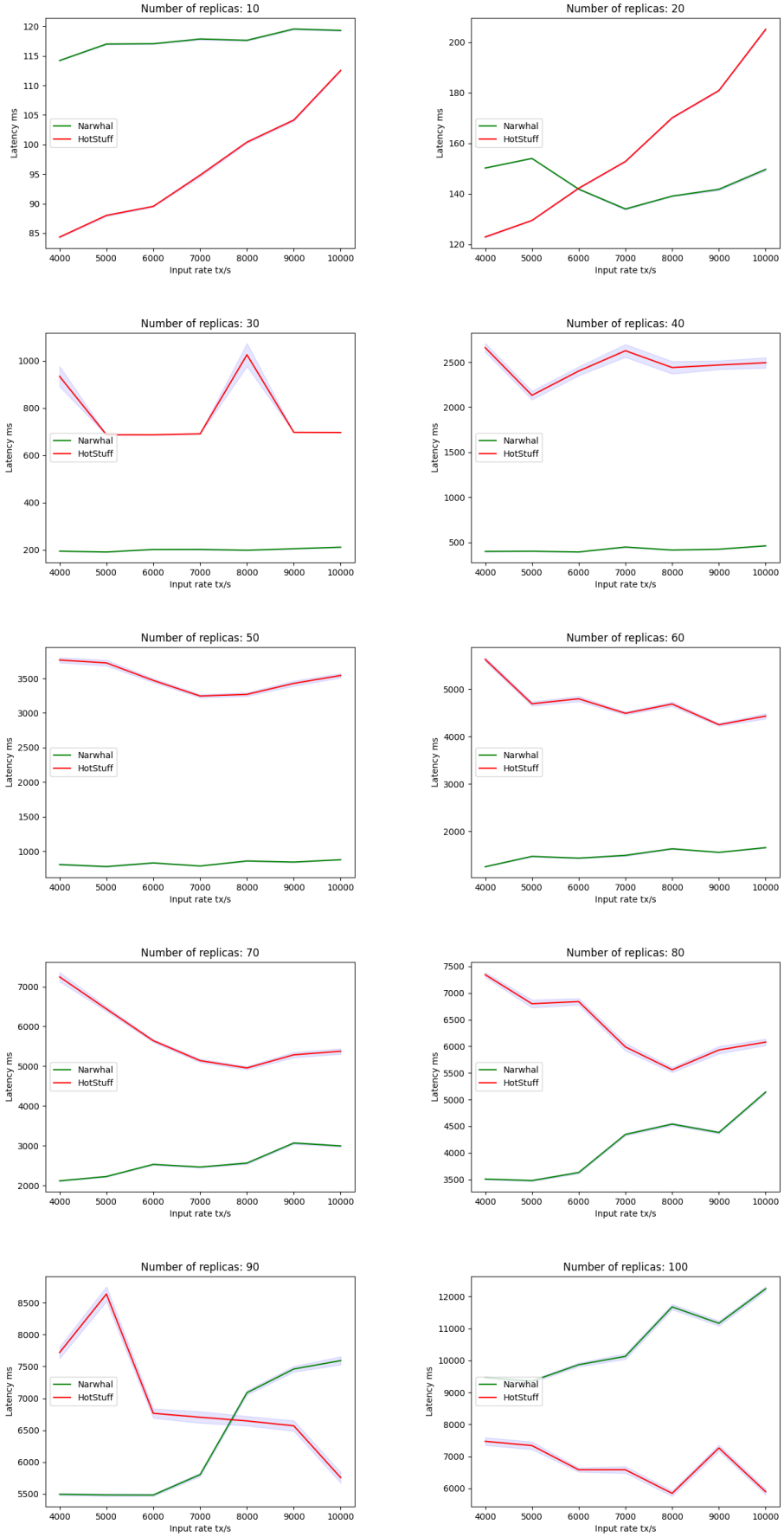


Figure A.3.: The latency for Scenario 1

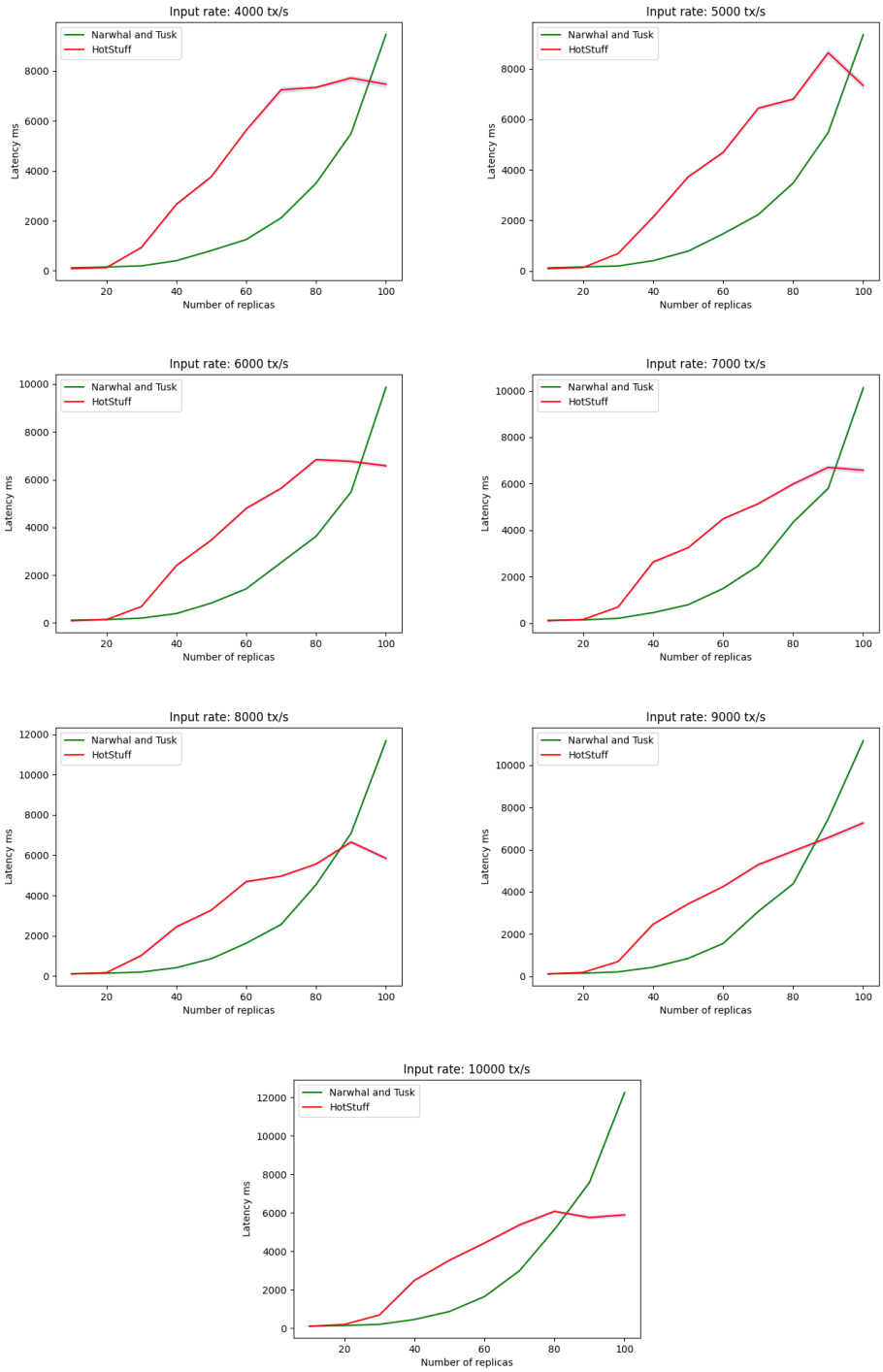


Figure A.4.: The latency for Scenario 1

B. Appendix 2: Complete Results for Scenario 2

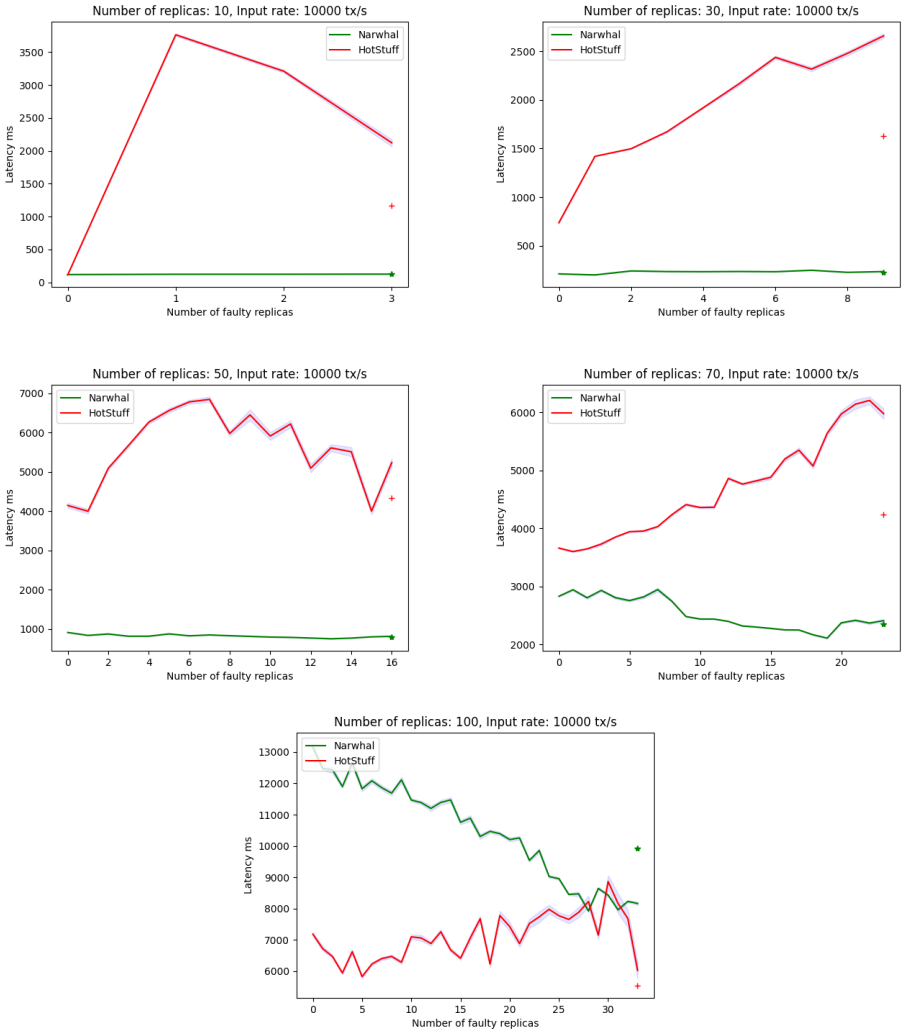


Figure B.1.: The latency for Scenario 2

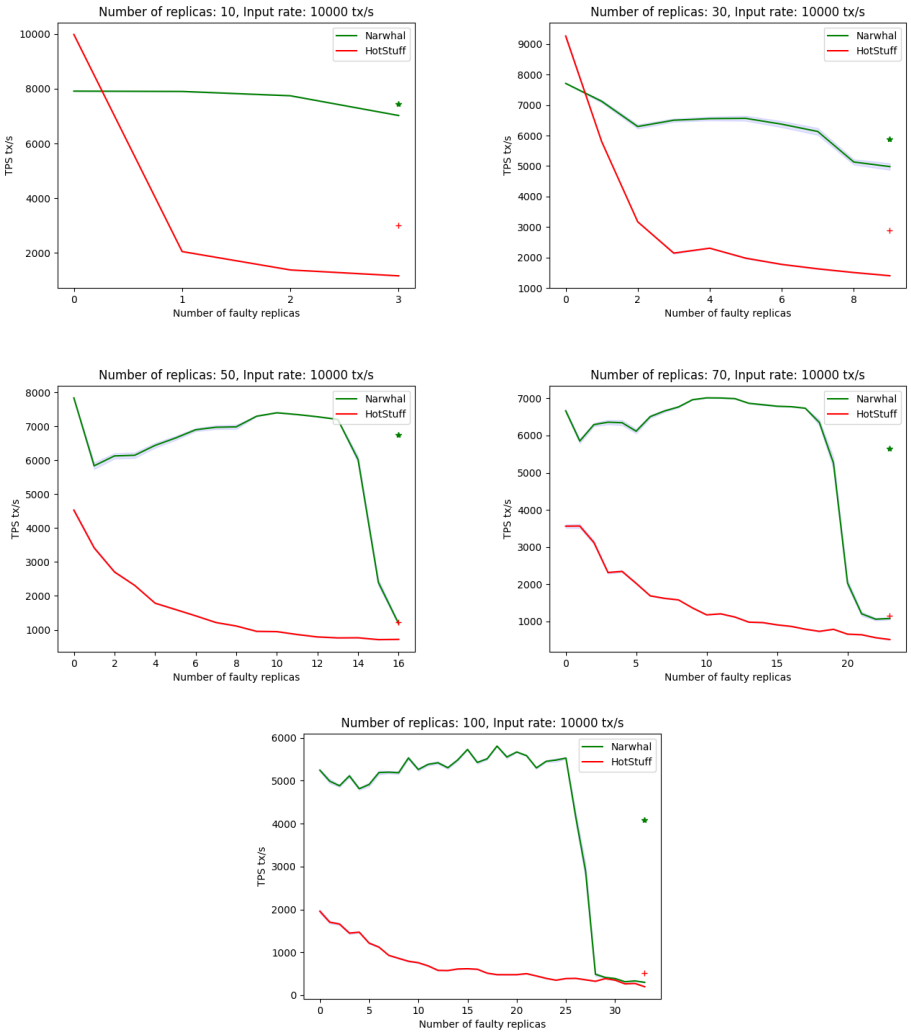


Figure B.2.: The TPS for Scenario 2

C. Appendix 3: Complete Results for Scenario 3

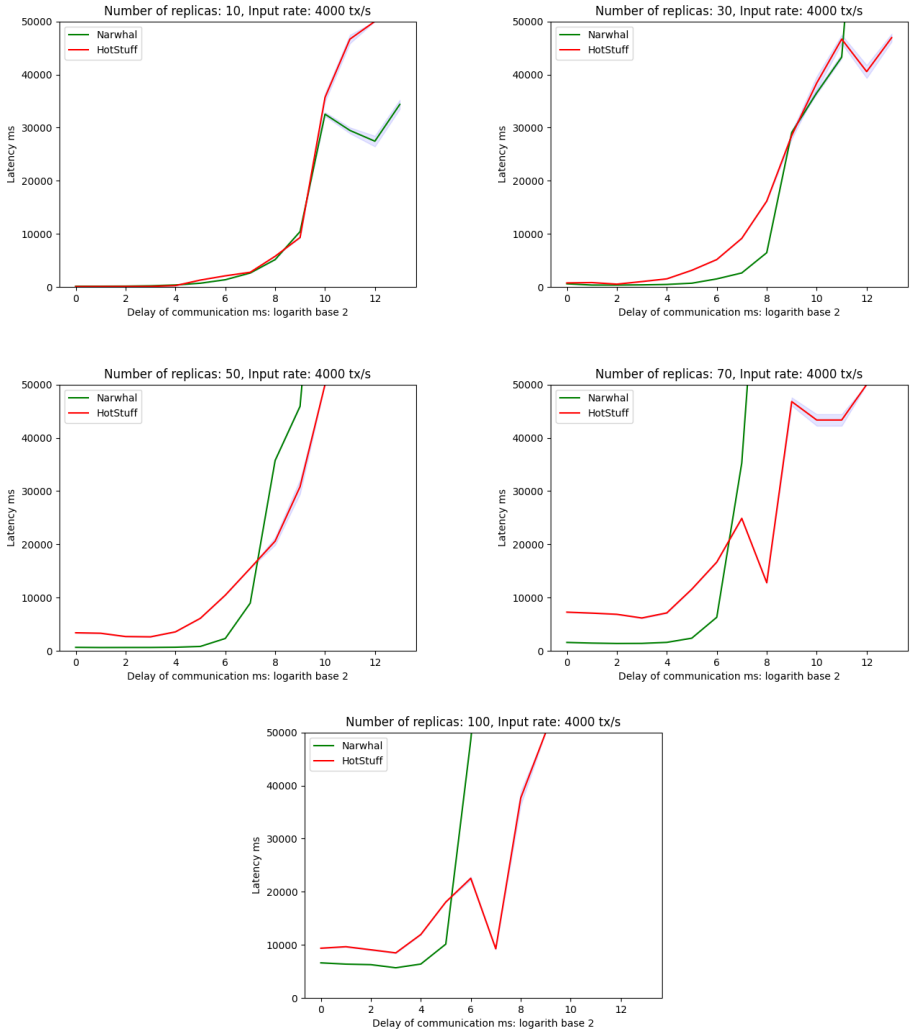


Figure C.1.: The latency for Scenario 3

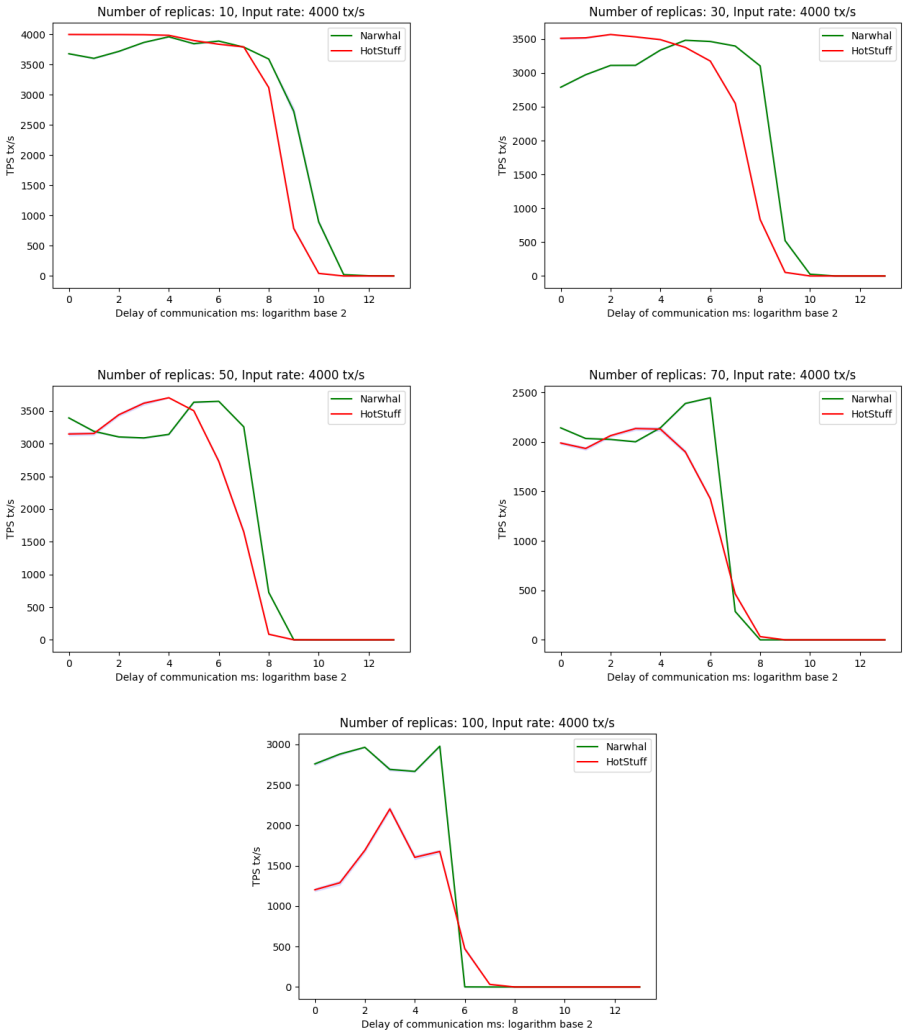


Figure C.2.: The TPS for Scenario 3