UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DETECTING AND MITIGATING FAULTS IN BYZANTINE FAULT TOLERANT SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Tuan Anh Tran

June 2024

The Dissertation of Tuan Anh Tran
is approved:

Peter Alvaro, Chair

Owen Arden

Alysson Bessani

Peter F. Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Detecting and Mitigating Faults in Byzantine Fault Tolerant Systems

by

Tuan Anh Tran

Byzantine fault tolerant state machine replication (BFT) is an approach to building highly available services that can tolerate any type of failure, including bugs in the software and adversarial replicas. The recent adoption of BFT protocols in blockchain systems has made what used to be primarily theoretical research practical at a large scale, and it has led to renewed interest in these protocols. Both classical (permissioned) BFT protocols and blockchain (permissionless) protocols share one common characteristic: they are designed to mask faults, i.e., they ignore faulty replicas and rely on correct replicas to keep the system functional. However, the existence of Byzantine faults in these systems is a real concern, and the failure to detect them can lead to violations of the correctness properties that BFT systems provide. For permissioned BFT systems, where strong consistency is prioritized in exchange for weak availability, and the number of replicas as well as the fault tolerance threshold are known and fixed, undetected faulty replicas can accumulate over time and eventually surpass the predefined fault threshold, after which the system can no longer guarantee availability (and possibly consistency). For blockchain systems, where strong availability is prioritized in exchange for weak consistency, an adversary can cause inconsistencies between a replica and the rest of the network. Therefore, these two classes of BFT systems present unique challenges in how fault detection can be used to strengthen their correctness properties.

In this dissertation, we investigate the major differences between permissioned and permissionless BFT systems and how they influence the behavior of faulty replicas. We recognize that although fault detection mechanisms exist in both of these systems, they are not used to mitigate any faults that arise. This work explores how we can enhance and utilize these fault detection mechanisms to make BFT systems more robust against Byzantine faults. For permissioned BFT systems, we introduce a novel reactive reconfiguration protocol, Phoenix, that integrates fault detection techniques that exist in literature with a reconfiguration mechanism, enabling a configuration manager to make informed reconfiguration decisions. By detecting and removing faults in these systems, we prevent them from accumulating and allow the services that run on these systems to be deployed for prolonged periods of time. For blockchain systems, we present mechanisms to detect and mitigate two major consistency attacks: eclipse attacks and execution fork attacks. Similar to our approach towards permissioned BFT systems, our fault detection mechanisms for these attacks solely rely on existing features within the blockchain, and our evaluation shows that they are effective in mitigating these attacks. Furthermore, we noticed that recent blockchain systems use smart contracts to provide automated membership management for a replica set but that the management mechanism cannot detect faulty replicas. We present Decentagram, a framework for highly-available decentralized messaging, with smart contracts that can authenticate digitally signed messages on-chain, paving the way for trustless, automated fault detection and reconfiguration.

To my sister, Elizabeth.

Acknowledgments

I would first like to acknowledge my advisors: Peter Alvaro and Owen Arden. Soon after I got my acceptance letter into the Ph.D. program at UCSC, Peter reached out to set up a meeting to discuss his and my research interests and invited me to attend one of his lectures. From that point on, I needed very little convincing to become a Banana Slug and join Peter's lab as one of his first Ph.D. students. Not only did Peter welcome me with open arms into his lab, but he also made sure that I would not have to worry about funding for the majority of my time at UCSC by vouching for me to be a recipient of, and ultimately being rewarded with, the Eugene V. Cota-Robles Fellowship. The one-on-one meetings we had while walking through the backwoods (and almost getting lost) or climbing a tree to sit on a makeshift seat built by other students will be one of the fondest memories that I will take with me.

At the beginning of my third year, I was lost in my research, with none of my projects materializing into publications. It was during this time that I took Owen's CMPS 223 course on advanced computer security, where I was introduced to a wide variety of topics delivered in the view of programming languages, and one of these topics, blockchains, became the focus of my research and a core part of this dissertation. What started with me catching Owen as he was heading back to his office and asking for a quick meeting to discuss research ideas eventually led to years of fruitful collaboration and mentorship that cumulated in several publications. For reasons still unknown to me, I do not know what initially motivated Owen to spend time every week listening to my crazy ideas and providing insightful feedback that helped guide me to become a better researcher, but for that, I am eternally grateful. Although he is a PL guy at

heart, I will never forget his recognition of just about every topic that is brought up, and his ability to always point me in the right direction.

In addition to my advisors, I would like to acknowledge the other members of my thesis committee: Cormac Flanagan and Alysson Bessani. I had the pleasure of taking Cormac's CMPS 203 course on programming languages during my first year, in which his interactive teaching style of using the whiteboard to deliver complex materials has influenced my own as a teaching assistant. Surprisingly, my most memorable interaction with him was not on campus but at a rock climbing gym, where I took a course to help overcome my fear of heights. Seeing one of your professors effortlessly tackle a huge bouldering wall gave me the much-needed motivation to finish that course. I also want to offer special thanks to Alysson, whose expertise in my field of research and thoughtful feedback have been crucial in shaping this dissertation. Despite having graduate students of his own to supervise and paper deadlines that he needed to meet, and despite the fact that we are on opposite ends of the globe where timezone differences are unforgiving, Alysson was more than happy to participate in both my qualifying exam and dissertation defense. Having "The Famous Alysson" (as one of my colleagues would say) on my committee and acknowledging my work gives credence to the research that I have done and that it could be helpful to others studying the same field.

I was lucky enough to have spent my time at UCSC surrounded by wonderful folks in the Disorderly Lab, Decent Lab, and LSD Lab. Roy Shadmon's knowledge of blockchain development has been invaluable in our work together, and his laidback approach to things and sense of humor provided much-needed comical relief during stressful times. I could always count on Priyanka Mondal for moral support when a paper submission gets rejected, or when

I need a helping hand in understanding a tough PL paper that has too many Greek letters. In particular, I would like to thank Haofan Zheng, who has been an amazing collaborator and whose insights and coding skills have been instrumental not only in getting our works published but also in producing high-quality artifacts for these works. Roy, Priyanka, and Haofan have been great colleagues and friends, and I am glad to have met them at UCSC.

I am also grateful to the researchers at Hewlett Packard Labs who provided me with the motivation I needed to start grad school. Particularly, I am incredibly thankful to Harumi Kuno, who took me under her wing as an intern at Labs when I was an undergrad and opened my eyes to the world of CS research. Before this, I was riding the high of my hubris, having tackled all the tough CS courses and thinking I was ready to end my journey in academia. My time at Labs was a humbling experience, and it made me realize how little I knew about the field I was studying. Harumi was a fantastic mentor who not only cared greatly about my progress as a mentee, but also my well-being as well as my family's.

Finally, I am extremely grateful for the support of my family and friends. My mother, who left everything behind to bring me to this country for a better opportunity in life. My father, who worked tirelessly so that I didn't have to survive on canned tuna and rice everyday during the toughest years in college. My sister, who has been my biggest supporter and my main motivation to finish grad school. And to all my friends at the Garage Hopes and Dreams, whose encouragement has helped make this dream a reality.

Chapter 1

Introduction

Modern distributed systems are complex, oftentimes involving a large number of loosely coupled components working together. Regardless of their complexity or deployment environment, we would like services built on these systems to be dependable. That is, even in the presence of faults, these services should remain available to process new operations; they are reliable, with limited to no interruptions for the lifetime of their deployment; their correctness properties are preserved; and they are maintainable, allowing for fault detection, mitigation, and recovery [130].

Byzantine fault tolerant state machine replication (BFT) is a well-studied approach to building highly available fault tolerant services that can tolerate any type of node failure, including compromised nodes, and can work in an asynchronous network environment. In this approach, operations that modify the service state are issued by clients and a set of replicas will execute these commands in the same sequence. As long as a minority fraction of these replicas are faulty at any single instance in time, the safety and liveness properties of the services are

guaranteed. However, the many protocols that resulted from years of research are either purely theoretical or have never made it past the proof-of-concept phase due to their complexity, assumptions of a powerful adversary that can coordinate corrupted replicas and network delivery, or poor performance with respect to crash fault tolerant protocols that considers no such adversary. In recent years, the advent of blockchain technology has led to renewed interest in BFT protocols, specifically the application of these protocols as the replication layer for decentralized services [14, 40, 219]. This has not only resulted in the development, deployment, and use of real-world BFT applications by millions of users around the world, but it has also revived research into making these protocols more practical and efficient.

Three major differences separate BFT protocols into two main classes. The first difference is in how these protocols consider membership of replicas that run consensus and execute client requests. Classical BFT protocols, such as PBFT [51], consider a closed and `permissioned` membership, where the set of replicas that participate in consensus is fixed, and the introduction of new replicas requires approval from some trusted source. In contrast, BFT protocols that are used in blockchain systems, such as Ethereum [213], consider an open and `permissionless` membership [158] in which any node can freely join the network and participate in consensus. The second major difference is in how BFT protocols make the fundamental tradeoffs between consistency, availability, and partition tolerance [38, 97]. Permissioned BFT systems [51, 64, 105, 72, 124, 39] prioritize consistency and partition tolerance, ensuring that the majority of the replicas are always in a consistent state with one another even if the network is partitioned, but that the service is only available during periods when the network is stable. Permissionless BFT systems [163, 213] prioritize availability and partition tolerance, allowing

the service to remain available in the presence of network partitions, but the state at subsets of the replicas may diverge, and that consensus decisions may be reverted even if they are decided by some of the replicas. Lastly, the difference in performance between these two classes of BFT protocols is significant, with permissioned BFT systems being able to provide throughputs that are often orders of magnitude higher than permissionless BFT systems. This is due to the fact that consensus in permissioned BFT systems involves a much smaller number of replicas, so consensus decisions can be decided at a faster rate.

In this dissertation, we investigate the ways in which these three major differences between permissioned and permissionless BFT systems influence the behavior of faulty replicas and how we can leverage fault detection mechanisms to improve the robustness of these systems against Byzantine faults. We recognize that one important weakness in both of these classes of BFT systems is the limitations of their fault detection mechanisms, or lack thereof, which can be utilized to the advantage of an adversary. For permissioned BFT systems, because the set of replicas is fixed, faulty replicas that are undetected can accumulate over time and eventually surpass the fault tolerance threshold, after which the system can no longer maintain its availability guarantees, and the service will halt. Although fault detection in these systems is a well-studied problem, all of the existing fault detection mechanisms ultimately leave the detected replicas in the active replica set. Furthermore, research has shown that reconfiguration is possible [142, 141, 31] but requires the existence of a trusted configuration manager with knowledge of which replicas in the active set are faulty. To this end, we present Phoenix, a reactive reconfiguration protocol for permissioned BFT systems that integrate existing fault detection mechanisms with reconfiguration to remove faulty replicas that have been detected by

correct replicas in the active replica set. By removing faulty replicas, we prevent them from accumulating over time and strengthen the availability guarantee of the service for prolonged deployments.

For permissionless blockchain systems, the possibility of consensus decisions being reverted creates an opportunity for an adversary to cause inconsistencies between a replica's view and the rest of the network. Due to the open nature of blockchain systems, each replica only knows about a subset of replicas in the network, and when new blocks are produced, they slowly propagate through the network via the replicas and their connections. An adversary that controls all of the connections to a replica can then launch an `eclipse attack`, where it can withhold new blocks from the replica and feed the replica with blocks that it has created, causing the blocks that the replica received from the attacker to be reverted once they sync up with the rest of the network. Current approaches to detecting eclipse attacks either rely on modifying the network layer to limit or reset peer connections, which either does not fully mitigate the problem or induces a high cost by requiring the replica to constantly send transactions to be confirmed by the rest of the network to verify that they are receiving data from the main chain. Similar to permissioned BFT systems, fault detection mechanisms exist in blockchain systems that can be used to detect the presence of an eclipse attack, but they are unutilized for this purpose. In order to determine that a block mined by a replica is valid, another replica can verify that the block's difficulty value, determined by the overall compute power of the network, is within a certain range. Since an attacker is assumed to have minority power in the network, they are unlikely to be able to produce blocks as fast as the rest of the network, so the difficulty in the blocks they produce will continually decrease. Our solution to detecting eclipse attacks uses the difficulty

value as a form of fault detection, and replicas can use our algorithm to monitor the difficulty of blocks that they receive over time to determine if they are being attacked. With a way to detect eclipse attacks, replicas in the network can be certain that their view of the blockchain is consistent with the rest of the network.

Another opportunity for an adversary to cause inconsistencies in blockchain systems is by exploiting the low throughput of the network, as described above. One of the major approaches that blockchain systems use to provide higher throughput is through off-chain scaling solutions, and of these approaches, state channels is one of the leading approaches that has seen significant research. To scale using state channels, parties will form a smart contract that encodes their state on-chain and then perform transactions with each other through direct communication off-chain that updates this state. At the time of their choosing, any party in the channel can submit an on-chain transaction that updates the on-chain state in the contract with the off-chain state that the parties have agreed on. However, due to the low throughput nature of the network, any high volume of transactions can cause network congestion, which can not only cause the on-chain update to be stuck but also increase the cost of sending these updates. An attacker that is a party to the state channel can exploit this by launching an `Execution Fork Attack`, in which they will update the state in the contract with a truncated off-chain state that benefits them. Current solutions to mitigating execution fork attacks do not consider the high cost of sending dispute transactions that an honest party must send to dispute the malicious state update. We present a solution to this problem by enhancing state channel contracts with a mechanism that will not only detect malicious state updates but also punish the attacker. Smart contracts in blockchains are rich in features, and state channel contracts already include the abil-

ity to verify state updates on-chain, but they ignore bad state updates when they are detected. Our `Robust Payment Channel` contracts consider the possibility of network congestion and require that any on-chain state update include a security deposit that will be forfeited and given to the party that submits a dispute that allows it to detect the malicious state update.

In our search for ways to enhance fault detection in blockchain systems and the resulting work in detecting eclipse attacks and execution fork attacks, we have found that smart contracts are proven to be a powerful tool for securing blockchain applications. Particularly, recent research [155, 184, 191] has shown that smart contracts can be used to manage membership of a replica group in permissioned blockchain system. However, these approaches require a single trusted entity to send authorized transactions that will trigger the replica replacement. This is reminiscent of an assumption made in reconfiguration in Phoenix (and other reconfiguration protocols such as [142, 31]), where a trusted configuration manager is required to make reconfiguration decisions. Ideally, we would like to adopt on-chain membership management for Phoenix in a way that would automate reconfiguration, allowing the system to have a reconfiguration manager without the need for trust. To this end, we will expand on the solution to the problem of on-chain fault detection by enhancing smart contracts with the ability to authenticate digitally signed off-chain messages to trigger on-chain application-specific logic. We present Decentagram, a framework for highly-available, decentralized publish-subscribe messaging. Our Decentagram smart contracts are able to authenticate signed messages from replicas to facilitate a voting scheme, detect equivocation, or detect replicas with compromised credentials that will cause the removal of faulty replicas. These contracts can be integrated with Phoenix to not only remove the assumed trust in the reconfiguration process but also make any

reconfiguration decisions highly available to all replicas and clients in the system.

1.1 Contributions

The contributions of this dissertation are as follows:

Fault Detection in Permissioned BFT: Reactive Reconfiguration [200]. We present the design, implementation, and evaluation of Phoenix, a reconfiguration protocol that allows permissioned BFT systems to detect and remove faulty replicas from the active replica set. Phoenix introduces a novel mechanism called reactive reconfiguration, which leverages the localized fault detection information from replicas to make better reconfiguration choices. The motivation behind Phoenix is that there is an abundance of fault detection techniques presented in literature, but as mentioned above, they do not provide a way to deal with faulty replicas even if they are detected. Additionally, we recognize that modern permissioned BFT systems already come with a reconfiguration mechanism that can replace replicas, allowing the service to be deployed for prolonged periods of time, but these mechanisms require the reconfiguration manager to know which replicas are faulty. By using fault detection to drive reconfiguration, we can detect faults as they occur and remove them before they accumulate over time and exceed the predefined fault threshold that the system can tolerate.

Fault Detection in Blockchain Systems: Detecting Eclipse Attacks [225]. We introduce a novel mechanism for detecting eclipse attacks in PoW blockchains that work by monitoring the difficulty value of new blocks. Our `Difficulty Monitor` improves on the current state-of-the-

art in that it is inexpensive since it does not require frequent on-chain checkpoint transactions or modification of the network stack. We noticed that blocks in a blockchain contain a rich amount of information, and the difficulty value that is a direct result of the underlying PoW consensus algorithm is already used by replicas to determine the validity of new blocks that they receive, but that replicas do not use this information for fault detection. The results of our evaluation against blocks in the Ethereum blockchain show that the `Difficulty Monitor` is effective and can detect eclipse attacks with negligible false positives and false negatives.

Fault Detection in Blockchain Systems: Detecting Execution Fork Attacks [202]. We present a solution to detect and mitigate execution fork attacks in blockchain systems that use state channels for off-chain scaling. Our `Robust Payment Channels` enhances payment channel contracts with a mechanism to detect malicious state updates and monetarily punishes the attacker by forfeiting their security deposit. In this work, we recognize that state channels are most vulnerable to execution fork attacks during periods of network congestion, and existing solutions do not consider the high cost of sending disputes for state updates that could result in execution forks.

Fault Detection in Blockchain Systems: On-chain authentication and Fault Detection. We introduce Decentagram, a framework for highly-available, decentralized publish-subscribe messaging. Decentagram smart contracts are able to authenticate digitally signed messages from replicas, allowing them to facilitate on-chain membership management by means of voting, equivocation detection, and leaked credentials detection. Our smart contracts provide a means for a system like Phoenix to encode the functionalities of its configuration manager on-

chain, removing the need for trust in the reconfiguration process. Additionally, Decentagram is the first system that supports instant on-chain cross-contract notifications, which can be used by applications that have multiple smart contracts that need to communicate with each other. Our evaluation of Decentagram shows that this approach is more efficient than state-of-the-art on-chain communication that uses the Monitor-and-React approach, which requires additional transactions to be sent on-chain. Furthermore, we will show that Decentagram can also be used to enhance our `Robust Payment Channels`, allowing them to automatically detect a malicious party that has been publicly exposed and preventing them from initiating a closing operation that can cause an execution fork.

1.2 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides the necessary background information on Byzantine fault tolerance state machine replication, the differences in correctness guarantees between permissioned BFT systems and blockchain systems, and the challenges to fault detection in both types of systems. Chapter 3 presents our reactive reconfiguration protocol, Phoenix, that leverages fault detection to drive reconfiguration in permissioned BFT systems. In Chapter 4, we present two variants of Phoenix, one that works under a synchronous network assumption, and the other which considers only crash-faulty machines. Chapter 5 describes our difficulty monitor for eclipse attack detection in PoW blockchain systems, including materials from [225]. The construction and implementation of the robust payment channel contracts that can mitigate execution fork attacks is presented in

Chapter 6, which includes material from [202]. Chapter 7 presents the Decentagram framework and discussions on its application to Phoenix and payment channels. Finally, Chapter 8 concludes this dissertation.

Chapter 2

Background

2.1 State Machine Replication

State machine replication (SMR) is an approach to building distributed systems that was first introduced in the foundational work of Lamport [139]. In this approach, a group of independent processes takes in a set of commands, and each process simulates a state machine by executing these commands with the goal of providing a total ordering on them [111]. If we want to use SMR to construct highly-available services, the underlying protocol must be able to tolerate faults [189]. Lamport subsequently introduced the Paxos protocol [138] in which a state machine transition is done after a subset of processes has reached consensus on the corresponding state machine command. By having a majority of the processes reach consensus on state machine commands, we can ensure that the total order on these commands is preserved even if a minority of the processes are faulty. The Paxos protocol was later revised to be easier understood [136] and has since resulted in many variants [137, 54, 114, 113, 140, 141], each

improving on a different aspect of the original protocol.

2.1.1 Byzantine Fault Tolerance State Machine Replication

SMR protocols usually consider two ways in which processes can be faulty: fail-stop (crash) and Byzantine (arbitrary). The failure model that Paxos and its variants support is crash failures, where a faulty process works correctly until a certain point and then halts, after which other processes no longer receive any messages from the crashed process. In yet another foundational work, Lamport introduced the concept of Byzantine faults [143], where processes can behave arbitrarily or even maliciously to prevent other processes from reaching consensus. While crash failures are much more frequent in the real world, non-crash faults have been shown to cause issues [21, 210, 162], which signifies the need for algorithms that can tolerate such faults. PBFT [51] was the first Byzantine Fault Tolerant SMR (or simply BFT) protocol shown to be able to not only provide both safety (agreement) and liveness (availability) in the presence of Byzantine faults but also be performant enough to be practical. Since the inception of PBFT, we have seen an abundance of research into these types of protocols, including directions that explore optimizing the communication pattern [154, 131, 76, 16, 165, 190, 102], relaxing network assumptions [160, 147], and applications to blockchains [219, 14, 12, 104, 115, 184, 146].

Byzantine Commit. BFT protocols consist of two subprotocols: a Byzantine commit protocol for normal-case execution and a view-change protocol to replace the leader when requests fail to move through the commit protocol. Figure 2.1 shows the communication pattern of a typical Byzantine commit protocol consisting of three phases. This type of Byzantine commit protocol was first introduced in PBFT and later adopted by other BFT protocols

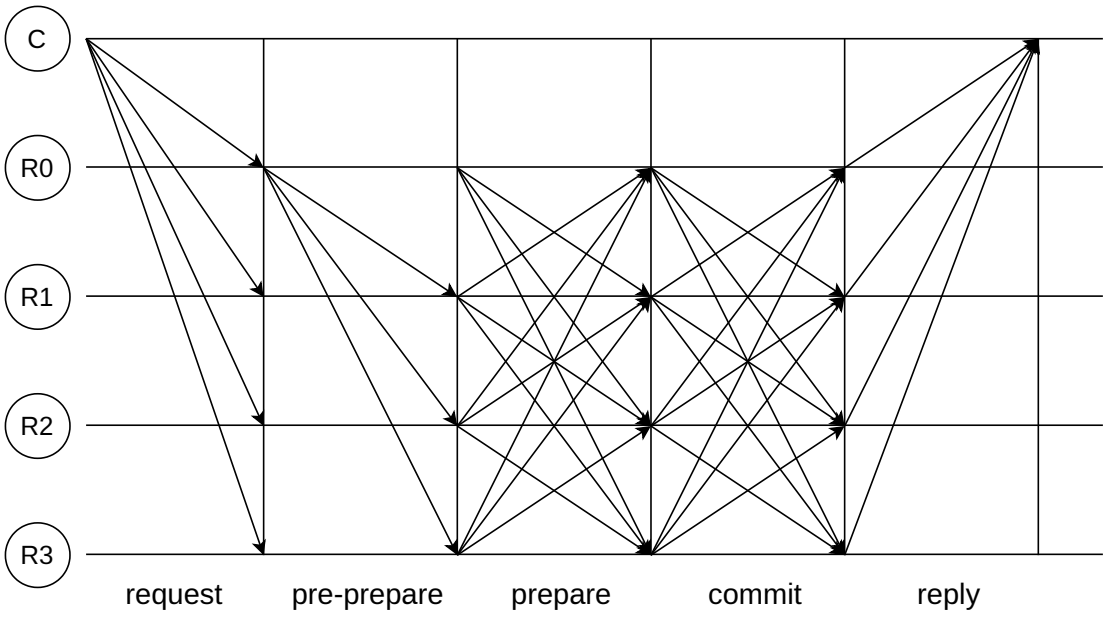


Figure 2.1: Byzantine commit protocol. A client sends a request to leader $R0$, which orders the request by proposing it in the next consensus instance. A replica commits and executes the operation once it receives a quorum of commit messages, and returns the result to the client.

[64, 31, 71, 13]. When a leader receives requests from clients, it uses *pre-prepare* messages to order these requests in a sequence and broadcast these messages to replicas. All replicas, including the leader, send *prepare* messages that acknowledge the validity of the *pre-prepare* to each other. A replica waits for a quorum of $2f + 1$ of these *prepare* messages to commit locally; a local commit at a replica ensures a total order of requests that it has seen. Replicas then exchange *commit* messages, and a quorum of these messages allows a replica to know that the request has been committed at enough replicas to execute it and reply to the client.

View Change. PBFT and similar protocols are leader-based protocols, meaning that a single process (we call replica) is responsible for proposing client requests in consensus instances identified by monotonically increasing sequence numbers, thus extending the totally

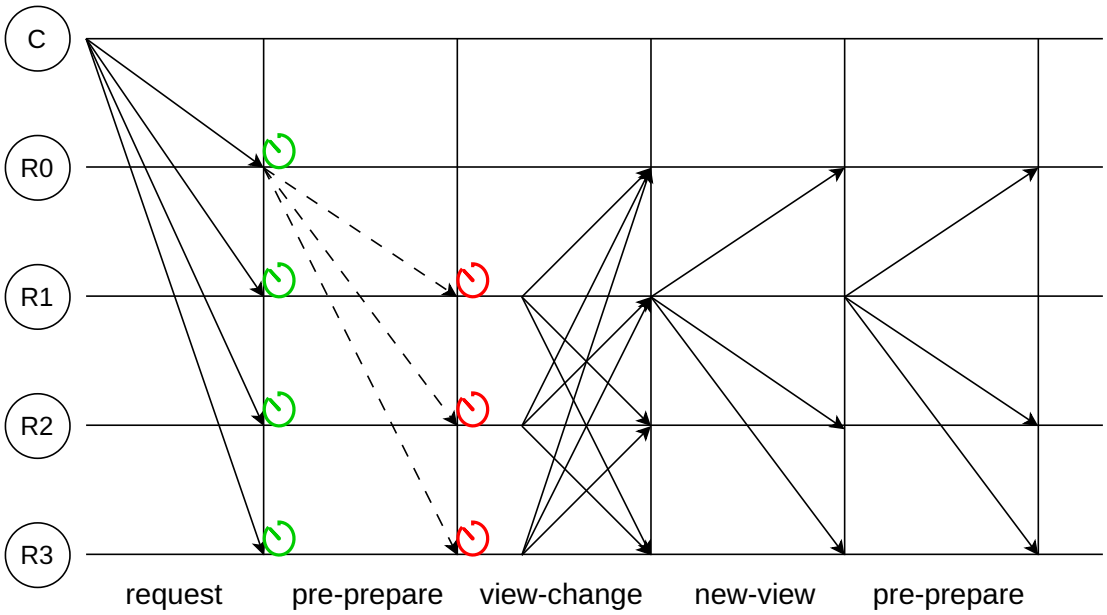


Figure 2.2: The view change protocol in PBFT. When replicas receive client requests, each replica will start a timer and wait for the *pre-prepare* message from $R0$ containing that request. If $R0$ does not send the message before the timer expires, the replicas will initiate a leader change to elect $R1$ as the next leader.

ordered log of committed requests with each successful consensus instance. Having a single replica propose requests in each round simplifies the design, but it makes the leader a single point of failure. For example, if the leader is Byzantine faulty, it could choose not to propose any client requests and cause the system to grind to a halt. The view change protocol solves this issue by electing a new replica as a leader if the current leader is suspected to be faulty.

Figure 2.2 shows the communication pattern for a standard view change protocol. In this figure, the replicas are currently waiting for requests, with replica $R0$ being the leader, and client C sends a request to be executed. At each replica is an internal timer that starts when they begin processing requests and stops when there are no more requests in the queue [51],

so the replicas will each start their timer after receiving the request from client C. The internal timer is key to the view change protocol because it allows replicas to elect a new leader when the timer expires, which happens when requests do not finish within the timer deadline. In this case, leader $R0$ is Byzantine and does not send a *pre-prepare* containing C's request, causing the timer to expire. When this happens, a replica initiates a leader election by sending a *view change* message to all the replicas, including the decision log containing consensus instances it has successfully executed, as well as consensus messages for instances that are known but have not finished. Leader election is a deterministic process where, given the current view number v, the leader of the next view $v+1$ is determined by $v+1$ modulo n. After collecting a quorum of *view change* messages with matching logs, $R1$ can consider itself successfully elected as the new leader and broadcast a *new view* message to all the replicas containing the set of logs that led to its election. Once a replica receives this *new view* message, it does the same verification steps on the logs before considering $R1$ as the new leader. With $R1$ now elected as the leader, it can start processing requests again, and it, along with $R2$ and $R3$, are able to drive consensus even if $R0$ remains faulty.

2.1.2 Reconfiguration

Reconfiguration is a procedure that changes the active replica set executing a BFT system [142]. Though this mechanism is seldom described in BFT papers, it is a necessary component in systems that are deployed for prolonged periods of time for two reasons. The first reason is that they can be used to add or remove replicas to change the degree of replication [168]. If the system is migrated to an environment where we wish to tolerate more simultane-

ous faults, the administrator can use reconfiguration to add more replicas to increase the fault threshold f. The second reason is that reconfiguration can be used to remove faulty replicas from the system. For each faulty replica that is removed, we essentially decrease the accumulated number of faults by one, and over time, could allow the system to tolerate an unbounded number of faults as long as no more than f fails at any one point [50]. However, determining which replicas are faulty is a challenging problem in BFT, especially when the network is asynchronous and consensus messages are not digitally signed. We will discuss in section 2.1.3 the challenges to fault detection and ways that state-of-the-art systems address them.

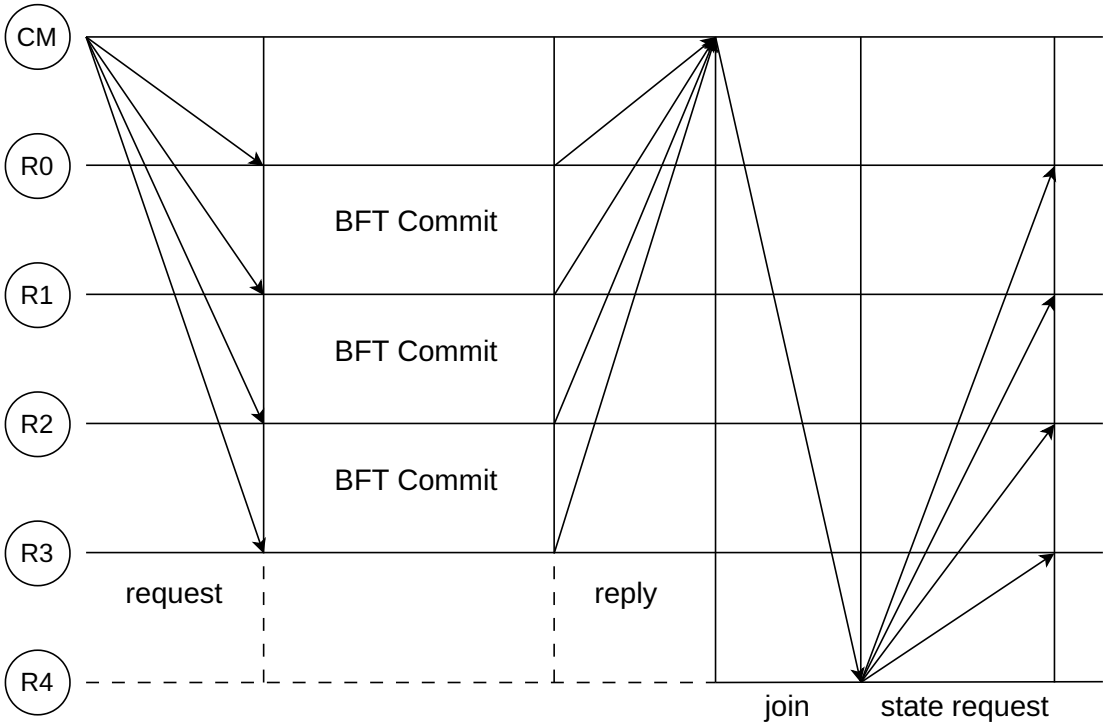


Figure 2.3: Reconfiguration in BFT-SMaRt. The configuration manager issues a reconfiguration request to replace $R3$ with $R4$.

At the time of writing, only two protocols have actually implemented a working re-

configuration protocol in a BFT system, BFT-SMaRt [194] and CCF [184]. Figure 2.3 shows the reconfiguration process for the BFT-SMaRt protocol. Here, a configuration manager (CM) process that is spawned by an administrator issues a reconfiguration request to remove faulty replica $R3$ and replace it with a backup replica $R4$. The CM first retrieves locally what it knows as the current active replica set, identified by their IDs. The CM then removes the ID of $R3$, then inserts the ID of the new replica $R4$. This newly constructed set will then be packaged into a client request and sent to the replicas (1). Replicas in the current active set will process this request just like any other client request, with the leader putting the request in a *pre-prepare* message, after which it goes through a consensus round (2). At each replica, a decision is reached once it has collected enough *commit* messages, then it will execute the request. A flag inside this special request will let the executing replica know to perform a reconfiguration operation, which involves changing the current active replica set to reflect the replica set within the request. It is assumed that the active replicas also know the IDs of the backup replicas and can connect with them once they are added. The replicas will then reply to the CM (3), who will wait to receive $f_B + 1$ replies containing replicas in the new view before considering the request as finished. If a new replica is added, the CM will notify the new replica with a message containing the new view (4); in this case, it will send the new view to $R4$. When $R4$ receives this message, it will first initiate a state transfer request from the other replicas in the new view it received (5) to catch up to the latest consensus decision before it participates in any future consensus instances.

2.1.3 Fault Detection

BFT systems are inherently masking quorum systems [152] designed to maintain consistency in the presence of faults by ensuring that any two quorums of replicas that commit a value intersect at enough correct replicas to preserve committed values and maintain a total order of these values. As long as the number of simultaneous Byzantine faults is less than the defined threshold, the remaining correct replicas will still be able to drive consensus without the safety property being violated, and the faulty replicas can be ignored [68]. The importance of preserving committed values is the reason research into BFT is more focused on the safety property [36]. Ideally, we would like to be able to detect faulty replicas as that would open up the possibility of removing them, but Byzantine fault detection is not a trivial task.

Approaches to fault detection in BFT systems are dependent on two aspects of the system model: message authentication and network assumption. With respect to message authentication, replica-to-replica messages, as well as messages between clients and replicas, can be authenticated using message authentication codes (MACs) [204] or public-key signatures [182]. Prior to PBFT, systems that were designed to tolerate Byzantine faults made use of digital signatures in the commit protocol [128, 181] in which a Byzantine replica could be detected if they had sent malformed messages. One optimization that made PBFT practical at the time was its use of MACs for consensus messages, but MACs do not provide non-repudiation, so one replica cannot prove that another is Byzantine by presenting messages that it has received from the faulty replica. Though the optimization was crucial at the time, recent work has shown that hardware advances and the use of multithreading can significantly reduce the performance gap

[31].

As for the network assumption, permissioned BFT systems typically assume a partially synchronous network [77], where the upper bounds on message delivery Δ and processor speed Φ are known but only hold after some unknown global stabilization time (GST). The reason why these systems assume this network model is because the FLP result [90] has shown that it is impossible to achieve consensus in an asynchronous network with faulty replicas, and the synchronous network model requires setting Δ to such a high value to prevent safety violations for the performance to be practical [53]. The challenge in detecting Byzantine faults in a partially synchronous network, outside of provably faulty behavior such as malformed messages signed with digital signatures, is that faulty behavior such as message omission or delays can be attributed to the network or the faulty replica, and there is no way to distinguish between the two. One example of this is when a Byzantine leader withholds *pre-prepare* messages, preventing replicas from committing client requests. Although replicas can suspect that the leader is faulty once their request timer expires and initiates a view change protocol to elect a new leader, correct replicas cannot be sure that the leader is faulty since the cause could have been a network partition.

The Configuration Manager. The configuration manager (CM) is an entity that helps the system recover by replacing faulty replicas, and it has connections with all the replicas in the current replica set as well as an additional set of spare replicas that are used as replacements. The existence of the CM is not unique to BFT systems, with CFT systems such as Raft [168], Paxos [141], and ZooKeeper [117] also relying on a CM to make reconfiguration decisions. However, the CM, as described in literature, makes two important assumptions.

First, there is the assumption of trust, wherein only a trusted process (i.e., an administrator) is allowed to issue reconfiguration requests to add or remove replicas. This trust is needed to prevent regular processes from issuing unnecessary, or even malicious, reconfiguration requests which would impact the performance of the system. In a BFT system where trust cannot be placed on any single process, it is preferable to have a trustless CM, and our work in automating fault detection, presented in Section 7 shows how we can encode the CM as a smart contract to remove the need for trust in the CM. The second assumption made about the CM is that it is expected to know which replicas in the active replica group are faulty when performing a replica replacement. Although this is a valid assumption in a CFT system, wherein the CM can have some assurance of a replica being faulty through a heartbeat mechanism, it is not the case for BFT systems since a Byzantine replica can communicate normally with the CM and abnormally with the other replicas.

This CM, as described in our work, serves three purposes. First, it can be used to modify the degree of replication by adding or removing replicas. This is useful for when replicas need to be taken offline for maintenance or added to increase f_B. Second, it can serve as an arbiter for faults in the system, taking in votes from replicas against other replicas they think are faulty; when there are enough votes against a replica, it can then remove the replica. We will describe in a later section how to leverage this functionality to serve as an imperfect fault detector to remove faulty replicas. By "imperfect," we mean that replicas can use the voting protocol to conclude that another replica is faulty, but occasionally, this conclusion could be false if the network partitions the correct replicas for prolonged periods of time. Lastly, in the synchronous version of our protocol described in Section 4, the CM can help the system recover

even when there are only $f_B + 1$ correct replicas remaining. Messages that are sent from the CM are signed, so replicas can trust these messages if the signature is valid. The version of the CM that we built for Phoenix is an extension of the CM in BFT-SMaRt (i.e., a trusted process).

2.2 Blockchain and BFT

One of the major differences that separates permissioned and permissionless BFT systems is how they behave under network partitions. The famous CAP theorem [97] states that there is a fundamental tradeoff between consistency, availability, and partition tolerance for any system that aims to solve distributed consensus. In such a system, we must choose between two of the three properties that we wish to preserve, and it is impossible to guarantee all three in an asynchronous network. Permissioned BFT systems guarantee consistency as long as the number of faults is within some bound, even if the network is partitioned, but is only available during periods when the network is stable (i.e., after GST). In contrast, permissionless BFT systems must always be available because of the constantly changing set of participants, any of which could take part in consensus and help produce blocks [103]. Additionally, since the participants can join and leave at any time, these BFT systems must also be able to tolerate a high degree of network partitioning. This comes at a cost, however, that participants can have an inconsistent view if they are partitioned from the majority for a prolonged period of time. Therefore, permissionless BFT makes the CAP tradeoff of sacrificing consistency in exchange for availability and partition tolerance. From now on, we will refer to permissionless BFT as simply blockchains. We will discuss in more detail how the possibility of inconsistent views of

the blockchain makes them vulnerable to attacks in Section 2.2.4.

2.2.1 Blockchain Consensus for Transaction Ordering

In blockchain systems, there are generally three types of participants: miners, non-mining clients, and users. Miners are those who participate in consensus by producing blocks that contain user transactions. These blocks are appended one after another, forming a chain with the goal that every participant in the blockchain network sees the same version of the chain, the main chain. Non-mining clients (or just clients hereafter) are those who wish to have their own copy of the blockchain locally but do not participate in consensus. Having a local copy of the chain is useful for validating new blocks seen on the network. Lastly, users are those who simply wish to transact on the blockchain, making up the majority of the network participants. These users could run their own clients through which they can use to send transactions, or they could use a third-party blockchain service [118].

Many of the popular blockchains that exist today (e.g., Bitcoin, Litecoin, Dogecoin, Monero) uses a consensus protocol called Proof-of-Work (PoW), which was introduced in the original Bitcoin paper [163]. In PoW, a miner that has packaged transactions into a block must do some computational work to show that the block is valid to others in the network. The work is done by computing a hash containing the root of the Merkle tree for the transactions in that block [157], the previous block's hash, and a nonce. The miner will try different nonces until the resulting hash meets a value called the *difficulty target*, a hash that is prefixed with a certain number of zeros. A block with a valid hash is considered to be *mined* and is then broadcasted over the network to be verified by other miners. After each block (or set of blocks),

the difficulty target adjusts depending on how fast the block was mined, with the goal of keeping the average mining time for blocks within some *block arrival time*. Note that although the process of mining a block is computationally intensive, verifying the mined block only takes one hash calculation. Recent research has shown that PoW-based blockchains are wasteful, requiring not only expensive equipment but also a large amount of energy to produce blocks [217]. Work on eliminating the waste in PoW has led to another class of blockchains called Proof-of-Stake (PoS) [44, 129, 187, 167]. In PoS, blocks are produced by validators instead of miners, and each miner is equally likely to be selected as the producer of a block. When a new block is produced by a validator, other validators will cast a vote on that block to place a weight on it, and the chain with the most weight will be the one considered by the rest of the network.

Blocks in the chain also have a notion of finality, that is, how many blocks are appended after a block is mined before the block is considered to be canonical. It can be the case that two miners mine two separate blocks that have the same parent block, resulting in a *fork* in the chain. Miners can choose to either discard their block or continue mining new blocks on top of their branch of the fork. The network rule is that the longest chain is considered to be the main chain, so eventually, one of these chains will win, and the parent can then be considered finalized. Because of the possibility of forks, blockchains are considered to be probabilistic state machines [186], meaning that each new block has a probability of being a block in the main chain. The notion of finality, or how many blocks *confirmations* are needed before the probability of a block not being part of the main chain is negligible, depends on the specifics of the underlying algorithm. For example, blocks in Bitcoin can be considered final after six blocks, whereas blocks in Ethereum can be considered final after 12 blocks [94].

2.2.2 Threat Model

When considering an adversary in BFT systems like PBFT, we are interested in the number of replicas that the adversary can control, the total of which cannot surpass f. In PoW blockchain systems, we are less interested in the number of machines but rather in how much computing resources these machines have. Recall that the PoW algorithm requires a miner to compute hashes in order to successfully mine a block, so the more resources a miner controls, the faster the miner will be able to find a valid hash. We assume an attacker physically and effectively controls no more than some percentage q of the total network hash power, Q. The PoW algorithm guarantees that it is computationally infeasible for an adversary that controls a minority hash power ($Q < 50\%$) to produce blocks at a rate fast enough to shift the chain to an inconsistent state. For PoS blockchains, we assume the attacker controls at most a fraction of the validators tolerated by the underlying protocol. In Ethereum, this is no more than $\frac{1}{3}$ of staked ETH (319k validators) to prevent halting progress and no more than $\frac{1}{2}$ of staked ETH (478k validators) to fork the blockchain or force a reorganization.

2.2.3 Scaling Blockchains

Leading cryptocurrencies such as Bitcoin and Ethereum are known to have scalability issues where the network fees explode when there is a large, sudden increase in usage [58, 32]. The bottleneck in transaction throughput comes from two limitations: the block size and block arrival time. In BFT, client requests can be *batched* [48] to amortize the communication cost of consensus, and the system administrator can configure the batch size to limit the number of requests per round. This is similar to the concept of blocksize in blockchains, where each trans-

action that is sent takes up a certain amount of memory to store the details of the transaction, and the protocol defines a limit on the maximum size each block can have, thus limiting the number of transactions that can go in each block. Then there is the block arrival time, which places a limit on how many blocks the network can produce regardless of the total amount of computing resources. These two limitations together restrict the total number of transactions that a blockchain can process per second, so during periods of usage with a large number of transactions, these transactions end up in what's called a *mempool* (or transaction pool) [112] to be processed later.

There are two approaches to scaling a blockchain to accommodate increased usage: On-chain scaling and off-chain scaling. On-chain scaling aims to improve transaction processing capabilities through changes in the protocol, such as increasing the block size to pack more transactions per block [1], improving the storage layer by sharding the database [209, 149], making the communication between nodes more efficient [169], and changing the consensus layer [96, 44]. However, on-chain scaling can lead to debate and fractures among communities [135, 127] and can take a long time to reach deployment, so off-chain scaling, which requires no changes to the protocol, is the preferred solution.

Some blockchains allow for smart contracts, self-executing code that is deployed on the blockchain, which users (or other contracts) can interact with. Applications can use these contracts to move data and computation off-chain, after which off-chain transactions can be made without incurring high on-chain transaction fees and latency. This approach to scaling blockchains is called off-chain scaling or layer-2 scaling. State channels [159], Plasma [173], and Rollups [43] are three major off-chain scaling solutions that utilize smart contracts.

2.2.4 Attacks that Violates Consistency

Even though it is not possible for the main chain to reach an inconsistent state if the majority of miners are honest, it is possible to corrupt a client's view of the chain by partitioning it away from the rest of the network. We now discuss two well-known and important attacks in blockchains that could arise from network partitions and why it is important that we are able to detect and mitigate these attacks.

2.2.4.1 Eclipse Attacks

At its core, blockchains are unstructured peer-to-peer networks where nodes work with little coordination. There is, however, a requirement that each node has a number of connections with other nodes in order to synchronize its view of the chain as well as propagate blocks [70]. Castro et al. [49] described an attack in peer-to-peer networks where an adversary can monopolize all connections to a node, effectively partitioning it away from the rest of the network and leaving it vulnerable to denial of service attacks. This attack is called an *eclipse attack*, a term that was coined in a later paper [192]. The eclipse attack was later shown to be effective in the Bitcoin network [110], where an attacker can eclipse both miners and non-miners, causing them to lose money.

One of the well-known consequences of an eclipse attack on miners that causes them to lose mining time and, in turn, money is the selfish mining attack [88]. Figure 2.4 shows an example of a selfish mining attack. Here, the adversary has eclipsed the honest miner, controlling all of the peer connections that it has to the rest of the network. Both the miner and adversary see the latest state of the main chain, with the latest block known as block i. The adversary and

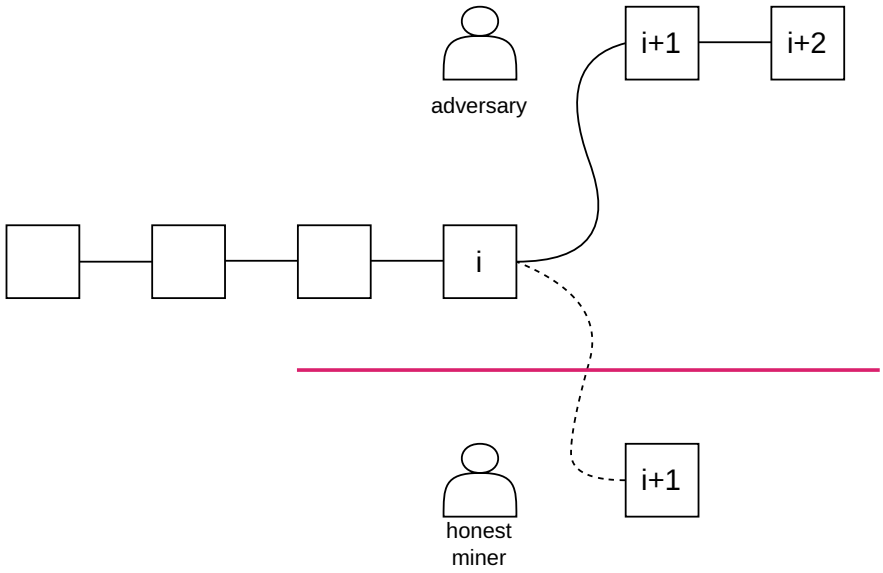


Figure 2.4: An adversary that has eclipsed an honest miner performs a selfish mining attack by causing that miner to waste computation on mining block $i+1$.

the honest miner mines block $i+1$ at the same time, but since the honest miner is eclipsed, the adversary does not relay either block to the network. Later on, the adversary mines block $i+2$ on top of its own block and then releases both its blocks and the honest miner's block. The network rule is to mine on the longest chain, so other miners will choose the adversary's blocks over the honest miner due to it being longer, leaving the honest miner to have to discard its block and not only lose out on the block reward but also the energy cost to mine its own block.

Non-miners are also vulnerable to eclipse attacks that can cause them to lose not only money but potentially material goods as well. Figure 2.5 shows an example where an adversary wants to use the same funds to pay for goods from two different merchants, and the adversary happens to be successful in eclipsing merchant1's client node. First, the adversary sends transaction *T1* to merchant1, who verifies the transaction before releasing the goods. The

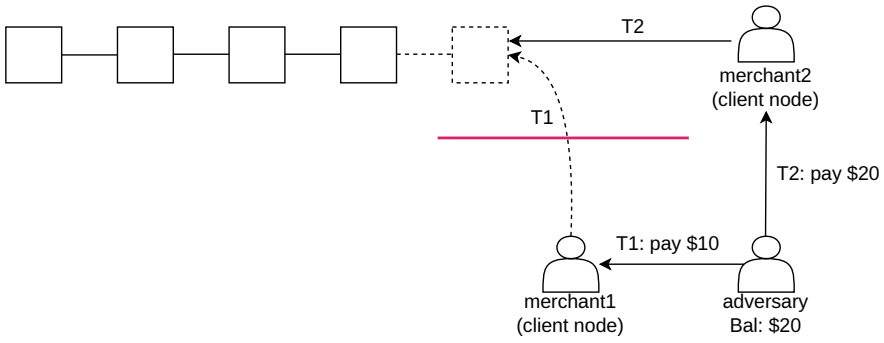


Figure 2.5: An adversary that has eclipsed merchant1 sends transaction *T1* to pay for goods, after which it sends a double spending transaction *T2* to merchant2 to pay for something else.

merchant sends *T1* over the network to be included in the next block, but the adversary intercepts and withholds that transaction. Then, the adversary sends transaction *T2* to merchant2 to pay for a different set of goods, which it receives after the transaction was verified. Since merchant2 is not eclipsed, it is able to successfully relay *T2* to the network, putting it in the next block and crediting its account with the adversary's funds. Even if the adversary stops eclipsing merchant1's node, the merchant can never claim the funds anymore since the transaction is now invalid due to insufficient funds.

2.2.4.2 Execution Fork Attacks

Payment Channels are instantiations of state channels that restricts participants to simple monetary transactions. In a payment channel, two users, Alice and Bob, who wish to transact off-chain can deploy a smart contract that contains a deposit from each of them. After creating the contract, Alice sends a message of the form (i, bal_A, bal_B, σA) to send a payment to Bob. In this message, i denotes the round number for the transaction, where a higher round number signifies a more recent transaction. Both Alice and Bob's balances after the payment

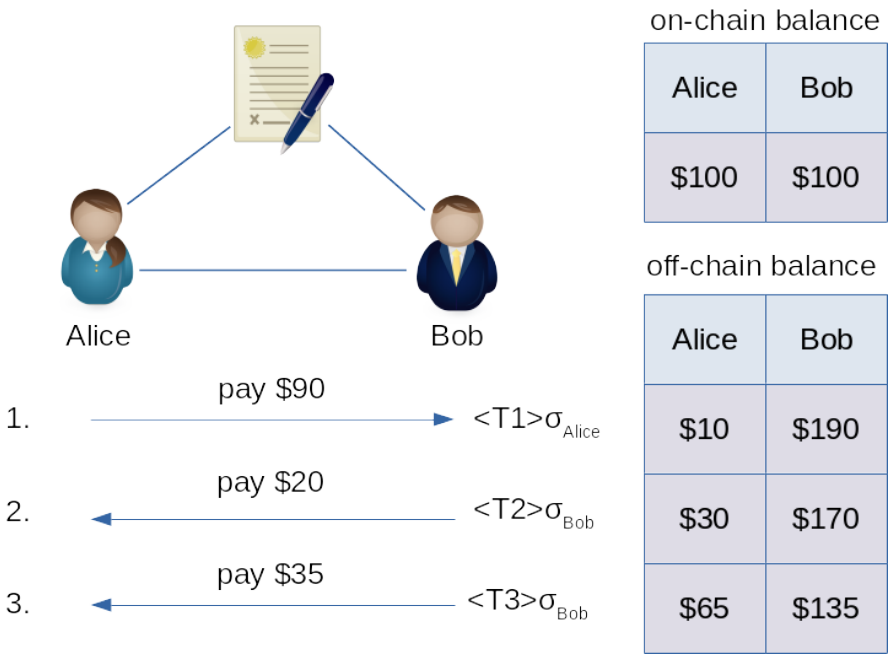


Figure 2.6: A bi-directional payment channel between Alice and Bob

are reflected by the fields bal_A and bal_B, respectively. Alice also includes a signature σA that shows her commitment to the round and user balances for that round.

If Bob wants to claim the money that is reflected by the balances in round i, he includes his own signature in the message to form (i, bal_A, bal_B, σA, σB) and send it in a transaction to the contract. The contract first verifies that the message has a higher round number than the last one it has seen, then verifies the signatures σA and σB. Upon successful verification, the contract will update the on-chain balances to reflect the off-chain balances in round i. After this, Bob can initiate a withdrawal of the amount in his balance from the contract but will need to wait for a predefined period of time (Δ_T). A withdrawal event will be signaled by the contract, and if there is no transaction from Alice after Δ_T has passed, Bob can withdraw his money.

Consider the example shown in Figure 2.6. Alice and Bob set up a payment channel

between each other, both committing to a deposit of $100 on the smart contract. Off-chain, both parties send a series of transactions (*T1*, *T2*, and *T3*) to each other. Under a typical network condition, Bob may issue a withdrawal request based on off-chain state *T3* with a transaction fee of $20. And as a result, he will get $115 back into his account. Bob realizes that when the network is under a period of congestion, the transaction fee may balloon to $70. In such a situation, if he withdraws the contract balance with the off-chain message *T1*, he will get $120 back to his account, which is $5 more compared to playing honestly. Alice may choose to submit a dispute saying that *T3* is the latest so that she can get $65 back. However, in order to submit this dispute, Alice has to pay $70 for the transaction fee. That means Alice not only gets nothing back to her account, but she will also lose another $5. If she keeps silent, she will actually get $10 back into her account when the network congestion is gone. Therefore, Alice is more likely to keep silent and take $10 back.

The action from Bob is a type of attack on payment channels called an execution fork attack [156]. Previous works have aimed to address one problem from these attacks, which is guaranteeing that Alice's dispute transaction makes it to the contract. We have shown with the previous example that Alice stands to lose money regardless of whether a dispute is successful, so ensuring the delivery of the dispute transaction only solves part of the problem. The smart contract needs a way to prevent Bob from making a malicious withdrawal when the network is congested, and there should be some way to punish Bob and compensate Alice if she is able to successfully dispute.

Chapter 3

Fault Detection in Permissioned BFT: Reactive Reconfiguration

Protocols like PBFT [51] require a configuration of $n = 3f + 1$ replicas, up to f of which can fail simultaneously without affecting the liveness or safety guarantees. There are several limitations to systems implementing these protocols. First, these systems achieve fault tolerance by masking faults, and faulty behaviors exhibited by one replica to another are simply ignored; the limited use of fault detection means that faulty replicas remain in the system indefinitely. Second, most BFT protocols are configured with a static set of replicas and do not explicitly support the addition or removal of replicas [30], meaning that even if faulty nodes are detected, no mechanism is available to remove them. If left unchecked, this would lead to an accumulation of faults that exceed the assumed upper bound (f) the system is capable of tolerating. Finally, if accumulated faults do eventually exceed f, Byzantine replicas completely control the availability of the system since there are now insufficient replicas to commit values

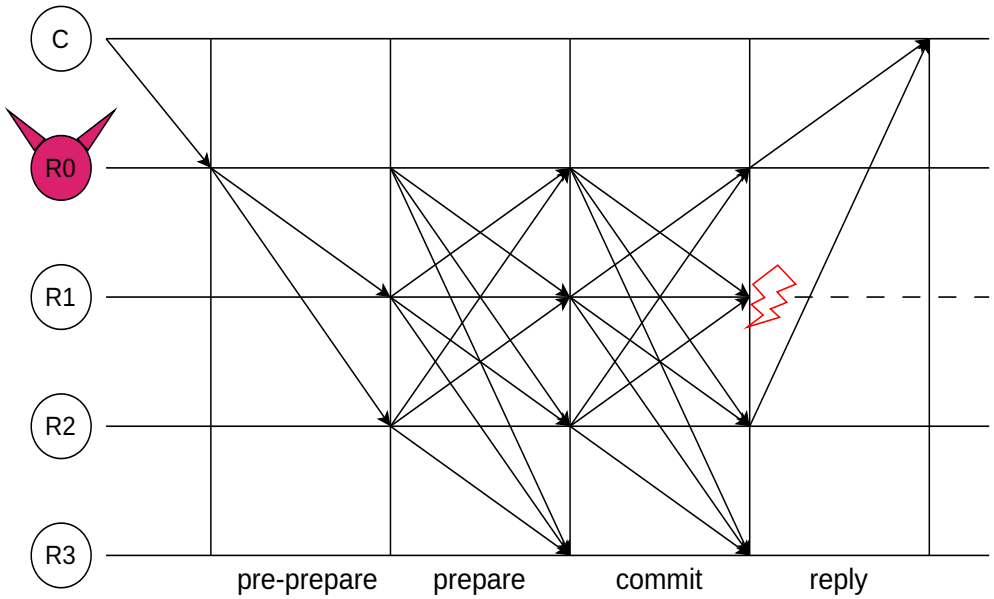


Figure 3.1: Client C issues a request to be executed, Byzantine leader $R0$ excludes $R3$ from the consensus instance to replicate the request, $R1$ crashes, and $R0$ stops sending messages and halts the system.

or elect a new leader. Worse, an opportunistic Byzantine node can simply behave honestly until f faults have accumulated, so the system will fail to maintain its correctness properties when it deviates from the protocol.

Figure 3.1 highlights this scenario for $f = 1$ (thus $n = 4$). In this configuration, committing a request or electing a new leader requires a quorum of $n - f = 3$ replicas to reach consensus. All replicas start at the same (orange) state, and a client issues a request to the current leader, $R0$. $R0$, which is Byzantine, excludes one of the replicas, $R3$, from the consensus round that processes the request and commits the request using only the remaining replicas. When all three have executed the request, replica $R1$ crashes, but replies from the two remaining replicas are sufficient to assure the client the request was successful. Once $R0$ suspects that $R1$ has crashed, however, $R0$ can choose to halt the system by not processing any additional

requests. A new leader cannot be elected because only two other replicas are functional. $R0$ can block any quorum attempting to make progress, so the system is stuck.

In this paper, we present Phoenix, a reconfiguration protocol that complements BFT protocols, allowing them to replace faulty replicas from the active replica set. Phoenix uses built-in fault detection mechanisms of BFT protocols to facilitate a voting scheme where replicas send votes to a configuration manager indicating which replicas they consider to be faulty. In order to tolerate an additional f_C crash faults over f_B Byzantine faults (with $f_C \leq f_B$), Phoenix requires $n = 3f_B + f_C + 1$ total replicas, commit quorums of size $n - f_B$, and reconfiguration quorums of size $n - f_B - f_C$. The configuration manager can then safely reconfigure the system as long as $n - f_B - f_C$ correct replicas remain (enough for a reconfiguration quorum). Note that tolerating the same number of faults in a traditional BFT protocol would require $n = 3(f_B + f_C) + 1$ replicas and quorum sizes of $2(f_B + f_C) + 1$.

We present an evaluation of two design choices regarding the *reply quorum* size or the number of replies a client must wait for before considering a request committed. Clients of protocols such as PBFT wait for a minimum of $f_B + 1$ replies so that at least one correct node has a commit certificate (which is required before a replica responds to the client) so that the committed request persists even in the event of a view change. In Phoenix, we must guarantee that clients do not consider a request committed until it is ensured to be preserved across reconfigurations. We evaluate two alternative design choices that meet this requirement. One design preserves the reply quorum size of $f_B + 1$, but requires replicas to digitally sign their *prepare* messages and construct a `prepare certificate` ($Cert_P$). With a reply quorum of size $f_B + 1$, $n - f_B$ of these replicas will have a $Cert_P$, so in a reconfiguration requiring a quorum of

$n - f_B - f_C$, at least one correct replica with $Cert_P$ will be able to prove its consensus decisions. The other design requires the client to wait for $n - f_B$ replies, but that replicas must sign their `commit` messages to form a `commit certificate` ($Cert_C$) By waiting for $n - f_B$ replies, this ensures that at least $n - f_B$ replicas have $Cert_C$, and during reconfiguration, at least one correct replica will be included in the reconfiguration quorum. Our results show that under reasonable latency assumptions, the benefits of the smaller reply quorum are significantly outweighed by the overhead of signing and verifying *prepare* messages.

3.1 System Model

Typically, the fault allowance f in traditional BFT protocols accounts for both crash and Byzantine faulty replicas. In our work of Phoenix and Sync Phoenix, we differentiate between crash faults and Byzantine faults and bound these faults by f_C and f_B, respectively. To avoid confusion, we will refer to the fault bounds of prior BFT protocols as f_B instead of f since Byzantine faults subsume crash faults in these protocols. Furthermore, we distinguish *commit quorums*, quorums that commit new entries to the log, from *view change quorums*, quorums that elect a new leader. To recover the system, we must first be able to remove faulty replicas and replace them with backups, then preserve the most recent state from correct replicas across reconfigurations.

We consider a distributed system consisting of a set of $n > 3f_B + f_C$ replicas and a set of client machines, all connected by an asynchronous network. Both clients and replicas can exhibit Byzantine faults and at any point in time, there can be at most $f_B = \lfloor \frac{n-1}{3} \rfloor$ Byzantine-

faulty replicas and an additional $f_C \leq f_B$ crash-faulty replicas.

3.2 Why Reactive Reconfiguration?

In section 2.1.2, we discussed the need for replication protocols to come equipped with a reconfiguration mechanism in order for them to be practical for real-world deployments. Reconfiguration enables systems to replace faulty replicas and dynamically change the active replica set, allowing a system to adjust its level of fault tolerance as needed. Then, in section 2.1.3, we mentioned that although BFT systems are masking quorum systems by design, approaches to fault detection are possible depending on the different aspects of the system model. In fact, fault detection is a well-researched topic in BFT, with many protocols providing different ways to detect faulty replicas by examining replica behavior during consensus. The motivation behind Phoenix is that these fault detection mechanisms are not utilized to their fullest potential because faulty replicas that are detected ultimately stay in the active replica set. However, if these fault detection mechanisms can be used to help replace faulty replicas through reconfiguration, then they can turn the rudimentary reconfiguration mechanism into a much more sophisticated and useful tool. Before we dive into the details of Phoenix, we first discuss why reactive reconfiguration is an ideal solution, and why reconfiguration and fault detection is not useful when working individually.

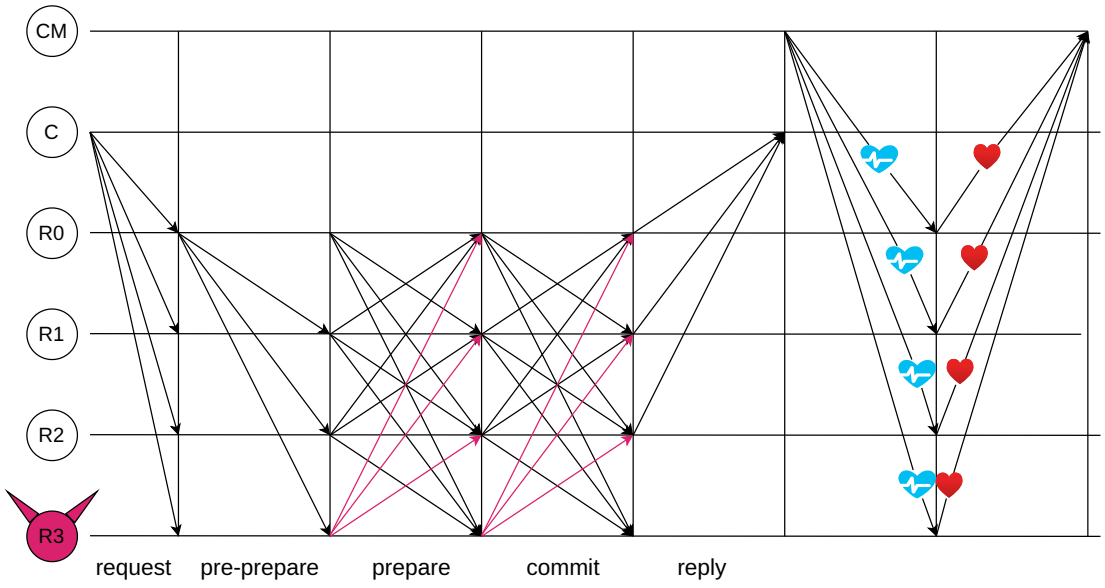


Figure 3.2: The configuration manager issues a heartbeat message and receives responses from all replicas, including Byzantine replica $R3$ who sends invalid messages during consensus.

3.2.1 Reconfiguration without Fault Detection

Reconfiguration, as shown in Figure 2.3, starts with the CM determining the change in the active replica set and then sending a reconfiguration request so that replicas can reconfigure to change their view to reflect this new set. One problem with reconfiguration, as described here, is that the CM must be able to determine which replica in the current set is faulty so that it can reconfigure to replace the faulty replica. This problem stems from the fact that the CM is not part of the active replica set, so it does not have all the necessary information to detect faulty replicas effectively.

Determining Faulty Replicas Naturally, in order to make reconfiguration requests, the CM must have a way to communicate with the replicas so there is a secure communication channel

between itself and each of the replicas in the active replica set. It does not, however, have access to the communication channels that the replicas establish with each other. More specifically, it does not see any of the replica-to-replica communication that happens during normal operation, the period of time in which the system is most vulnerable to Byzantine-faulty behavior. This means that the only form of fault detection that the CM can perform to determine which replicas to remove is heartbeat messages using its communication channels with the replicas. But, the possibility of a faulty replica being Byzantine, in addition to partitions in the network complicate even this limited form of fault detection.

Consider the example in Figure 3.2. A client sends a quest, which gets ordered by the current leader $R0$, and successfully goes through a round of consensus. One of the replicas, $R3$, is Byzantine and sends invalid messages during the *prepare* and *commit* phases, but consensus is nonetheless successful, and the client receives enough replies from the replicas. The only participants that are aware of this behavior are the replicas since they received the messages from $R3$ and discarded them due to their invalidity, and the fault is masked from the client since it sees that the request was successful.

During normal operations, the CM will perform fault detection by occasionally sending heartbeat messages to check whether the replicas are still alive, with one round of these messages making it to the replicas after the client receives the replies. Since it does not want to be removed from the active replica set, $R3$ will respond to this request in a timely manner, leading the CM to conclude that all replicas are alive and well and that no reconfiguration is necessary. This is not ideal since $R3$ could continue sending invalid messages to other replicas indefinitely, essentially being a useless replica. With non-participation, *R3* also becomes

an accumulated fault that could push the number of faults in the system over the threshold if other replicas were to fail. If the CM had access to the consensus messages that were sent, it would have been able to reach the same conclusion as the replicas, but including the CM as a recipient to all consensus messages not only adds unnecessary complexity to the protocol, it also impacts the performance due to replicas having to send more messages with each consensus round. Furthermore, even if that was implemented, *R3* could just send valid messages to the CM but invalid messages to the other replicas, and the other replicas cannot prove that the messages originated from *R3* if digital signatures are not used.

3.2.2 Fault Detection without Reconfiguration

Fault Detection Techniques Earlier in Section 2.1.3, we discussed how the aspects of a BFT protocol, such as the network assumption and the use of signatures, affect fault detection. We now discuss what popular BFT protocols consider as faulty behavior in replicas and how these behaviors can be detected.

- **Throughput Monitoring.** Every consensus instance starts at the primary, who has control over the construction of *pre-prepare* messages. The rate at which consensus instances succeed determines the throughput of the system, and there are two ways that a malicious leader can affect the throughput. The first way is that the leader can intentionally slow down the rate at which it sends these *pre-prepare* messages. Internally, replicas rely on timers to trigger leader elections, with request timers ensuring that client requests eventually finish (preserving the liveness property). However, this does not prevent a primary from delaying the inclusion of a request in a *pre-prepare* message until near the expiration

of the timer, which could take place long after the request was received. In fact, in BFT-SMaRt, this timer, by default, is set to a generous 2 seconds to account for the possibility of a slow network. A Byzantine primary can leverage this mechanism (that exists as a means to replace a faulty primary such as itself) to its advantage and selectively increase the latency for clients by choosing when to include their requests in a consensus instance.

The second way that a primary can affect the throughput is by limiting the number of requests that it includes in a batch for a consensus instance. Batching is a well-known optimization that is used to improve throughput in protocols that provide linearizability [91], wherein multiple client requests can be grouped together in a single request batch, and replicas reach consensus on the batch instead of individual requests. Typically, an implementation will define a maximum batch size, as a batch that is too large takes longer to process due to the amount of time needed to send over the network as well as the time needed to process each request in the batch [188]. Though systems can define a maximum batch size, there are no minimum batch size requirements since requests can arrive at a slow rate. Thus, a primary could choose to construct batches that will process the client requests before the corresponding timer expires, but the batch sizes are small enough that throughput is nowhere near optimal.

Replicas in Aardvark [64] detect this behavior by using a throughput monitor that keeps track of the number of requests that have successfully passed through consensus and executed. Moreover, replicas will continually ”raise the bar” by increasing the expected number of requests that should be executed, and the primary will have to meet the throughput

target each time; otherwise, the primary will be replaced. Though this will increase the number of leader rotations as the throughput will eventually reach a point that either exceeds a replica's processing power or the network's bandwidth, it prevents a malicious primary from intentionally slowing down the system.

- **Message Flooding** When a replica receives a consensus message from another replica, it first checks whether the message is valid by verifying its contents and signature (or MAC). The message is accepted if it is valid and the replica has not already considered the message for the current phase of consensus. Recall that in the *prepare* and *commit* phases, a replica can only move forward if it has received messages from a quorum of replicas, and any single replica's message only gets accounted for once. Practical BFT protocols are designed to work under an asynchronous network assumption so messages sent can be duplicated by the network. The way to deal with duplicate messages is to simply discard them since it could be a faulty network that has sent the message multiple times.

  At any point in time, a replica can receive many messages, so they need to be stored somewhere to be processed. Usually, this is done using a queue or multiple queues, one for each of the replicas that are connected. A malicious replica can make use of this by duplicating messages to fill up the queue of another replica. Not only does this waste resources of the correct replica, but it could also delay the processing time of valid messages since all the invalid messages still need to be processed.

  The Aardvark protocol also presents a solution to this type of faulty behavior by having

replicas keep track of the rate of messages that they receive from other replicas. If the rate of messages coming from one replica exceeds the rates from other replicas by a certain threshold (20 times in Aardvark), then the replica is considered to be intentionally flooding the network.

- **Fairness in Request Processing** Most BFT protocols are leader-based protocols, where clients send their requests to who they know as the current leader. With the power to decide which requests get proposed in a consensus instance, a Byzantine primary can choose to include requests from certain clients at a much lower rate than other clients. This, in turn, places the throughput of any client at the mercy of the primary, who can essentially starve some clients while favoring requests from other clients.

  The way that Aardvark deals with this is by requiring that client requests be included in a round-robin order. Remember that clients also forward requests to all the replicas, so they all know of the requests that have been sent by clients. Each replica, when executing requests in a batch proposed by the primary, keeps track of the most recent requests that it has executed from each client. By seeing the difference in the rate at which requests successfully execute differ from one client to another, replicas are able to detect whether a particular client is being starved and suspect the primary.

- **Inconsistent Logs** A primary can also misbehave by proposing two consensus instances with different requests for the same sequence number and have two sets of replicas reach agreement on these instances, thereby causing replicas to have conflicting views of what the log entry for that sequence number should be. This has been shown to cause problems

in past protocols, where a consensus instance that has been successfully committed is later replaced due to conflicting proposals [5, 6]. Of course, this is the worst that can happen as it violates the consistency property that BFT protocols are designed to provide. Detecting conflicting proposals would require replicas to keep track of the proposals from the primary and exchange them with each other, which can be done in a view change.

During a view change, replicas will broadcast a *ViewChange* message to all the replicas, and the primary of the next view is elected if it receives a quorum of these messages. Within a *ViewChange* message, a replica includes its *commit* log, which contains the sets of *commit* messages for each consensus decision that was successful. Here, a replica can also include its *prepare* log, and replicas would be able to compare this log with the *commit* log to determine whether a malicious primary has issued conflicting proposals. XFT [147] uses this approach to detect inconsistent logs caused by the primary, allowing replicas to broadcast their suspicion to others if a conflicting proposal is detected. Hotstuff [7] uses a similar strategy to detect primary equivocation by having replicas check for conflicting block proposals.

- **Invalid Consensus Messages** In the *prepare* and *commit* phases of consensus, replicas exchange messages with each other, and a quorum of these messages allows them to move forward to the next phase. Again, a consensus message is only accepted if its contents are valid and, in the case of view changes, the signature over that message is also valid. Earlier, we discussed how replicas simply discard duplicated messages, but we can keep track of the rate of duplicated messages to determine if a replica is intentionally flooding

the queue. In the case of invalid consensus messages, replicas also discard them by default since they cannot be considered in the quorum of messages needed to move on to the next phase. Instead of ignoring these invalid messages, we can also have replicas act on these messages before discarding them.

Replicas in PRRW [195] will raise suspicion if they receive such invalid messages from another replica. The suspicion is raised by sending out a *detect* message to other replicas, who will also suspect the replica if they have received *f+1* of these *detect* messages. Since replicas in BFT protocols typically rely on MACs instead of digital signatures for consensus messages for better performance, a quorum of *f+1 detect* messages ensure that at least one correct replica has received an invalid message from the suspected replica.

Systems	**Faulty Behavior**	**Detection**	**Action**
Aardvark [64]	Delay Propose Message, Reduce Batch Size	Throughput Monitor	View Change
Aardvark [64]	Neglecting Clients	Fairness Detection	View Change
Hotstuff [219]	Conflicting Proposals	Digital Signatures	No Commit
Aardvark [64]	Replica Message Flooding	Message Rate Monitor	Blacklist
XFT [147]	Inconsistent Logs	Digital Signatures	View Change
PRRW [195]	Invalid Messages	Content Check	Reboot

Table 3.1: Different types of faulty behavior that can be exhibited by both primary and non-primary replicas, how correct replicas can detect them, and the action taken upon a successful detection.

Action Taken after Successful Fault Detection Table 3.1 summarizes the different types of faulty behavior that can be exhibited by both primary and non-primary replicas and how BFT protocols can detect them. For faulty behaviors that are detected in the primary replica, such as limiting throughput, starving clients, or issuing conflicting proposals, the only course of

action is to remove the primary through a view change, as is done in Aardvark and XFT. In the case of faulty behavior in non-primary replicas, correct replicas in Aardvark will temporarily blacklist the offending replica for some time, while replicas in PRRW will trigger a reboot. What these protocols have shown is that faulty behavior exhibited by malicious replicas can be detected by making use of the rich information that is available to correct replicas and that actions can be taken to mitigate such behavior to make the protocol more robust. Nevertheless, these mitigations are imperfect as the offending replicas, even if truly malicious, ultimately remain in the active replica set. Without a way to remove them from the system, these detectably faulty replicas still serve as accumulated faults that will eventually surpass any fault threshold defined by the protocol.

3.3 Phoenix: Reactive Reconfiguration

As we have seen in the previous section, BFT literature already describes mechanisms for dynamic reconfiguration and novel fault detection techniques that cover a wide range of faulty behavior in both primary and non-primary replicas. However, these mechanisms by themselves are lacking since reconfiguration requires knowledge of replica interaction, and fault detection ultimately leaves the detected replicas in the active replica set. We now show how Phoenix integrates these two mechanisms, allowing us to build BFT systems that not only detect faulty replicas but also remove and replace them with new replicas.

3.3.1 Voting: Fault Detection

Using Phoenix, we would like to leverage the fault detection techniques described in Section 3.2.2 with minimal changes to the underlying BFT protocol. We first define the notion of subjective and objective faults, then present the voting protocol in Phoenix and describe how it removes replicas exhibiting these types of faulty behavior.

Subjective Faults. We define a subjective fault to be one that cannot be proven to a third party; examples of these include limiting throughput, starving clients, message flooding, and invalid messages, as described in Aardvark and PRRW in Section 3.2.2. These types of faults cannot be proven to a third party because they are locally detected at a single replica, which can be malicious and lie about their experiences. For example, for consensus messages that are only authenticated using MACs, a malicious replica can forge an invalid message and claim that it was received from another replica. Over time, a replica i that experiences these types of faulty behavior from another replica j might suspect j to be faulty, after which i will ask the CM to replace j. The CM can consider these requests as votes to remove the suspected replica, with a sufficient number of votes ($f_B + 1$) signifying that at least one correct replica considers the suspected replica to be faulty.

Objective Faults. For protocols such as XFT and Hotstuff, where digital signatures are used, and equivocation can be detected locally at a replica in the form of inconsistent logs and conflicting proposals, the voting protocol in Phoenix can be easily adapted. Because these faults can be proven to a third party, we define them as objective faults. Unlike subjective faults,

where $f_B + 1$ votes are needed to make sure that at least one correct replica considers a replica to be faulty, objective faults can be proven by verifying signed messages. A replica i can prove to another replica j that replica k is faulty by showing j two conflicting messages signed by k. Therefore, if j is a correct replica, it does not have to wait for $f_B + 1$ votes to consider k as faulty, and can immediately vote against k.

Voting Protocol. The voting protocol in Phoenix is shown in Algorithm 1. When a replica `k` experiences faulty behavior from another replica `j`, but the behavior is subjective, the replica can relay this suspicion to the CM and the other replicas. To do this, replica `k` will construct a message $\langle VOTE, j, Cert, \emptyset \rangle_{\sigma_k}$, where `j` is the suspected replica, `Cert` is the certificate for the latest consensus decision containing signed messages from a quorum of $n - f_B$ replicas, and Øis an empty proof. If, on the other hand, `k` experiences objectively faulty behavior from `j` (i.e., it has proof that `j` is faulty), then `k` will construct a similar message but with the proof included.

When replica `k` receives a message from another replica `i` containing a vote against replica `j` will first validate that both `i` and `j` are in the current configuration, and the certificate is valid. Validating the certificate consists of verifying that the certificate contains $n - f_B$ signed messages from distinct replicas in the current configuration. After the certificate is validated, the decision number of the consensus instance for the certificate is compared against the decision number of the latest consensus decision in the local log. If the decision number in the certificate is lower than that of the local log, it means that replica `i` is behind and needs to catch up. It could be the case that the decision number in the certificate is higher than that of `k`'s log, in which case `k` will need to request state from the other replicas to fill in the gaps in its log. Next, the

Algorithm 1 Voting Protocol

▷ ***as a replica k***

When replica `j` is suspected
  broadcast $\langle VOTE, j, getLatestDec(\texttt{DecLog}), \emptyset \rangle_{\sigma_k}$
When replica `j` is detected with Proof
  broadcast $\langle VOTE, j, getLatestDec(\texttt{DecLog}), Proof \rangle_{\sigma_k}$
When received m : $\langle VOTE, j, Cert, Proof \rangle_{\sigma_i}$ from replica `i`
  if `i` in `Conf` $\wedge$ `m.j` in `Conf` $\wedge$ isValidCert(`m.Cert`)
    if `m.Cert`.decNum $<$ getLatestDecNum(`DecLog`)
      send $\langle STATE\_TRANSFER, DecLog \rangle_{\sigma_k}$ to `i`
    else
      if `m.Cert`.decNum $>$ getLatestDecNum(`DecLog`)
        stateRequest()
      if (Proof $\neq \emptyset \wedge$ isValidProof(m.Proof)) $\vee$ VoteCount(`m.j`) $> f_B$
        broadcast $\langle VOTE, m.j, m.Cert, m.Proof \rangle_{\sigma_k}$
      else
        incrementVoteCount(`j`)
When received m : $\langle VOTE\_REQUEST, j, Cert, Proof \rangle_{\sigma_{CM}}$ from CM
  if `m.Cert`.decNum $>$ getLatestDecNum(`DecLog`)
    stateRequest()
  `Cert` $\leftarrow$ getLatestDec(`DecLog`)
  broadcast $\langle VOTE, m.j, m.Cert, m.Proof \rangle_{\sigma_k}$

▷ ***as the CM***

Upon init(R, B)
  `Conf` $\leftarrow R$
  `Backup` $\leftarrow B$
  latestDec $\leftarrow \emptyset$
When received m : $\langle VOTE, j, Cert, Proof \rangle_{\sigma_i}$ from replica `i`
  if `i` in `Conf` $\wedge$ `m.j` in `Conf` $\wedge$ isValidCert(`m.Cert`)
    if `m.Cert`.decNum $>$ latestDec.decNum
      if (m.Proof $\neq \emptyset \wedge$ isValidProof(m.Proof))
        broadcast $\langle VOTE\_REQUEST, m.j, m.Cert, m.Proof \rangle_{\sigma_{CM}}$
      else
        wait for $(f_B + 1)$ matching votes
        broadcast $\langle VOTE\_REQUEST, m.j, m.Cert, m.Proof \rangle_{\sigma_{CM}}$
      latestDec $\leftarrow$ `m.Cert`
    else if `m.Cert`.decNum $=$ latestDec.decNum
      wait for $(n - f_B - f_C)$ matching votes
      reconfigure(m.j, m.Cert)
    else
      send latestDec to `i`

proof contained in the vote message is checked. A valid proof will inform k that j is objectively faulty, so k can immediately send its own vote against j using the same proof provided by i. An empty proof will inform k that j is only suspected of being faulty, so it must wait for $f_B + 1$ votes before sending its own vote against j. This step ensures that if $f_B + 1$ replicas have committed to voting out j, then all correct replicas will eventually commit to voting out j. View changes rely on a similar approach to allow all correct replicas to send their own view change messages once they believe that a correct replica has committed to a view change (after receiving $f_B + 1$ *view-change* messages) [51, 131].

It is possible that more than one replica will be suspected by two distinct groups of $f_B + 1$ replicas, so the voting protocol must ensure that replicas will reach agreement on a single faulty replica to remove (the other suspected replicas can be removed in subsequent reconfigurations), and this replica can be determined by the CM. When the CM receives a vote message, it will first also check that both the replica sending the message and the replica being voted against are in the configuration and that the certificate contained in the message is valid. Then, the CM will either immediately broadcast a vote request or wait for $f_B + 1$ matching votes before sending the request, depending on whether the proof contained in the vote message exists and is valid. The CM will only reissue its vote request if it receives a subsequent vote with a higher decision number contained in the certificate, meaning that, at most, a single faulty replica will be chosen when $n - f_B - f_C$ replicas are up-to-date. A replica receiving a vote request from the CM will also vote against the replica specified in the request, even if it has already voted against another replica. By having the CM decide on the faulty replica to remove, the protocol guarantees that all correct replicas will eventually agree on the same faulty replica to remove

from the current configuration.

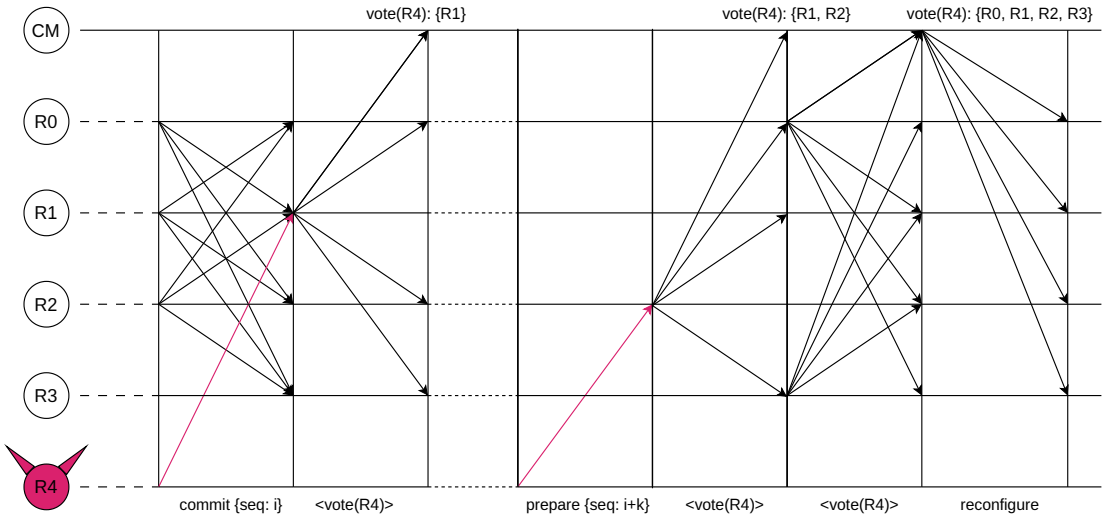


Figure 3.3: Replicas voting to remove a faulty replica in Phoenix.

Figure 3.3 shows an example of replicas voting to replace a faulty replica in Phoenix. In this example, there are five replicas in the current active replica set: R_0, R_1, R_2, R_3, and R_4 with R_4 being the Byzantine replica. After running for some time, the replicas reach consensus instance identified by sequence number i, and R_4 behaves maliciously towards R_1 by sending an invalid message during the *commit* phase. R_1 suspects R_4 to be faulty and broadcasts a *vote* message to the replicas and the CM. Then, in some future consensus instance $i+k$, R_4 sends an invalid *prepare* message to replica R_2, causing it to suspect and vote against R_4. Replica R_0 and R_3, upon receiving the vote from $R2$, now has f_B+1 votes against R_4 and will send their own votes against R_4. Once the CM receives the votes from either R_0 or R_2, it will have received $n-f_B-f_C$ votes against R_4, the threshold needed to safely initiate a reconfiguration. This threshold of $n-f_B-f_C$ ensures that a reconfiguration that involves communication between

$n - f_B$ replicas will intersect in at least one correct replica that has the consensus decision $i + k - 1$, the last successful consensus decision. A successful reconfiguration will thus guarantee that this value will propagate to $n - f_B$ replicas in the new configuration and prevent any subsequent consensus decisions from overwriting it.

3.3.2 Reconfiguration

The difficulty in reconfiguration is to make sure enough correct replicas commit to the new configuration. BFT-SMaRt does reconfiguration through the Byzantine commit path that is used to commit client requests, which requires $n - f_B$ replicas. If we do not consider the existence of additional f_C crashed replicas (i.e., $f_C = 0$), reconfiguration requests from the CM would eventually finish even if the Byzantine replicas do not participate since the remaining replicas form a sufficiently large CQ. Now, we must consider the case of $f_C > 0$ and how these additional crashes affect the decision of the reconfiguration request. If we were to set CQ at $n - f_B - f_C$ $(2f_B + 1)$, it would be easy to see that safety can be violated. Two CQs can intersect at only a Byzantine leader, which can equivocate, causing two groups of correct replicas to decide on different values for a single log entry. If, instead, we have CQ be $n - f_B$ $(2f_B + f_C + 1)$, then the Byzantine replicas can withhold messages when there are crashed replicas, and the reconfiguration request can never be completed. Thus, since we cannot achieve both safety and liveness, replicas have to be able to reconfigure through a different path other than the commit path. This separate path is the reconfiguration path that requires $n - f_B - f_C$ replicas.

Algorithm 2 describes the reconfiguration process in Phoenix. After the CM receives $n - f_B - f_C$ matching votes against a replica j, the `reconfigure` procedure will be invoked to

Algorithm 2 Reconfiguration Protocol

▷ ***as the CM***

procedure RECONFIGURE(j, Cert)
  replacement ← getReplacement(Backup)
  `newConf` ← `Conf` \ {j} ∪ {replacement}
  broadcast $\langle RECONFIG, Cert, \texttt{newConf} \rangle_{\sigma_{CM}}$
  wait for $(n - f_B - f_C)$ matching replies
    `reconfReply` ← {s : matchingMessage(REPLY, reconfDec)}
  `Conf` ← `newConf`
  `latestDec` ← reconfReply.reconfDec
  send $\langle JOIN, Cert, \texttt{Conf} \rangle_{\sigma_{CM}}$ to replacement

end procedure

▷ ***as a replica k***

When received m : $\langle RECONFIG, Cert, newConf \rangle_{\sigma_{CM}}$ from the CM
  broadcast $\langle SYNC, m, decLog \rangle_{\sigma_k}$
  wait for $(n - f_B - f_C)$ matching sync messages
    reconfDec ← {s : matchingMessage(SYNC, m, decLog)}
  decLog ← decLog ∪ reconfDec
  latestDec ← getLatestDecNum(decLog)
  send $\langle REPLY, latestDec \rangle_{\sigma_k}$ to the CM
  `Conf` ← `newConf`

When received m : $\langle SYNC, reconf, decLog \rangle_{\sigma_i}$ from replica `i`
  validate m.reconf
  validate m.decLog
  syncDecNum ← m.reconf.Cert.decNum
  if syncDecNum > getLatestDecNum(decLog)
    if syncDecNum = m.decLog.decNum
      decLog ← m.decLog
    else
      stateRequest()
  numSyncMsgs ← numSyncMsgs + 1

initiate the replacement of j. First, a replacement replica is selected from the set of backup replicas, and the configuration is updated to remove j and add the replacement. Next, the CM will broadcast a `reconfig` message to all the replicas in the current configuration, including the certificate for the latest consensus decision and the new configuration. The CM will wait for $n - f_B - f_C$ matching replies from the replicas confirming that they have agreed on the new configuration before updating the configuration and sending a `join` message to the replacement replica to inform them of their addition to the active replica set. The steps performed by the CM are similar to the steps in the reconfiguration protocol implemented by BFT-SMaRt [194], except for one important difference. In BFT-SMaRt, the reconfiguration procedure is initiated manually by an administrator with knowledge of the faulty replica j. This is not the case in Phoenix, as the procedure is automatically triggered after the CM receives enough votes against j by the other replicas.

When a replica k receives a `reconfig` message from the CM, it will first verify the signature on the message before broadcasting a `sync` message. The `sync` message includes the message from the CM as well as k's decision logs containing all the consensus decisions up to the latest one it knows about. Once k receives enough `sync` messages, it will extend its decision log to include a new consensus decision containing all the sync messages and then send a reply to the CM. A replica k receiving a `sync` message from replica i will first validate the contents of the message, verifying that the reconfiguration message within the message has the signature from the CM and that the included decision log is valid. If the decision number in the message from the CM is higher than the highest number in its decision log but equal to that of the one sent by i, then k can adopt the decision log from i, otherwise k is behind and will need to perform

a state request to catch up before proceeding with the reconfiguration.

3.4 Safety in Reconfiguration

Phoenix must preserve the following properties.

Property 1. A reconfiguration cannot be triggered by only the faulty replicas and must be requested by at least one correct replica.

Property 2. All correct replicas in the new configuration must start with the same state, and this state reflects the latest consensus decision from the previous configuration.

Property 3. Requests acknowledged by $n - f_B$ replicas must be preserved across reconfigurations.

Reconfiguration, like view changes, is costly and results in a temporary halt to the service as replicas stop processing messages in order to reconfigure. Property 1 prevents Byzantine replicas from being able to force the CM to initiate a reconfiguration, even if they collude. The CM must wait for at least $n - f_B - f_C$ *vote* messages before it replaces a replica.

Property 2 and a weaker variation of property 3 ($f_B + 1$ replies instead of $n - f_B$ replies) must be guaranteed by a standard view change algorithm [103]; therefore they also must hold for reconfigurations. To guarantee Property 2, the replica(s) with the latest state in the old configuration must be able to relay this state to the CM. This is done by having replicas send in the *Cert*, which contains the signed consensus messages for the latest consensus instance, in their `vote` message. A quorum of $n - f_B - f_C$ of these vote messages with matching *Cert* means

that at least one non-faulty replica has sent *Cert*, and it is the latest decided consensus decision (since $n - f_B$ signed messages are necessary to construct *Cert*). The CM can then relay *Cert* to all the replicas in the new configuration, ensuring that correct replicas in the new configuration start at the same state.

Property 3 allows clients to have a consistent view of the replica state. Clients must know when to consider their request as successful in order to move on to subsequent requests, and replies from correct replicas to the client must be consistent across reconfigurations. Previous designs of BFT allow a client to consider a request as successful after $f_B + 1$ replies from different replicas, but this is not sufficient when there are f_B and f_C simultaneous faults. There is a tradeoff between which consensus message to sign and the number of replies a client waits for before moving on. In BFT-SMaRt, the *commit* messages can be signed, which allows replicas to build certificates that can be verified. With this approach of signing messages in the last phase of consensus, clients have to wait for $n - f_B$ replies before they can move on. Waiting for $n - f_B$ replies from replicas when `commit` messages are signed means that a reconfiguration quorum of $n - f_B - f_C$ replicas will intersect this reply quorum in at least one correct replica with this decision, and this replica can prove it using the signed `commit` messages. PBFT-PK (PBFT with signatures [48]) relies on signatures on the *prepare* messages to prove consensus decisions in a view change. If signatures are used on these `prepare` messages, clients can wait for a smaller number of replies ($f_B + 1$) between requests. Although the reply quorum is smaller, $f_B + 1$ replies mean at least one non-Byzantine replica has replied to the client, and this replica has received $n - f_B$ `commit` messages before replying. These $n - f_B$ messages must have come from replicas that have, in turn, received $n - f_B$ signed `prepare` messages to create a certifi-

cate before sending the `commit` message. Of these $n - f_B$ replicas with this certificate, at least one correct replica will be included in the reconfiguration phase and can prove the consensus decision with the signed messages in the certificate. Based on our evaluation in section 3.5.1, adding signatures to the *write* phase would have a bigger impact on performance than waiting for more replies, so in Phoenix, we followed the latter approach.

3.5 Evaluation

In this section, we present our evaluation of Phoenix, which consists of two sets of experiments. The goal of the first experiment is to measure the performance overhead of the two design choices against the baseline off-the-shelf implementation of BFT-SMaRt. In the second experiment, we look into the latency of reconfiguration to see how fast a system is able to recover when there are crash and Byzantine faults.

3.5.1 Large Quorum vs. Write Signatures

In this first experiment, we provisioned a cluster of four replica machines and 2 clusters of four client machines on Chameleon [65]. Each machine contains two Intel Xeon Gold 6242 "Cascade Lake R" processors (16 cores @ 2.8GHz, 32 threads, 192GB RAM), all running Ubuntu18.04. A total of 2400 "closed-loop" client processes are launched across the eight client machines, all continuously issuing requests.

Figure 3.4 shows the throughput and latency of the three protocols (baseline, large reply quorum, *prepare* signatures). For each of these protocols, we evaluated their performance

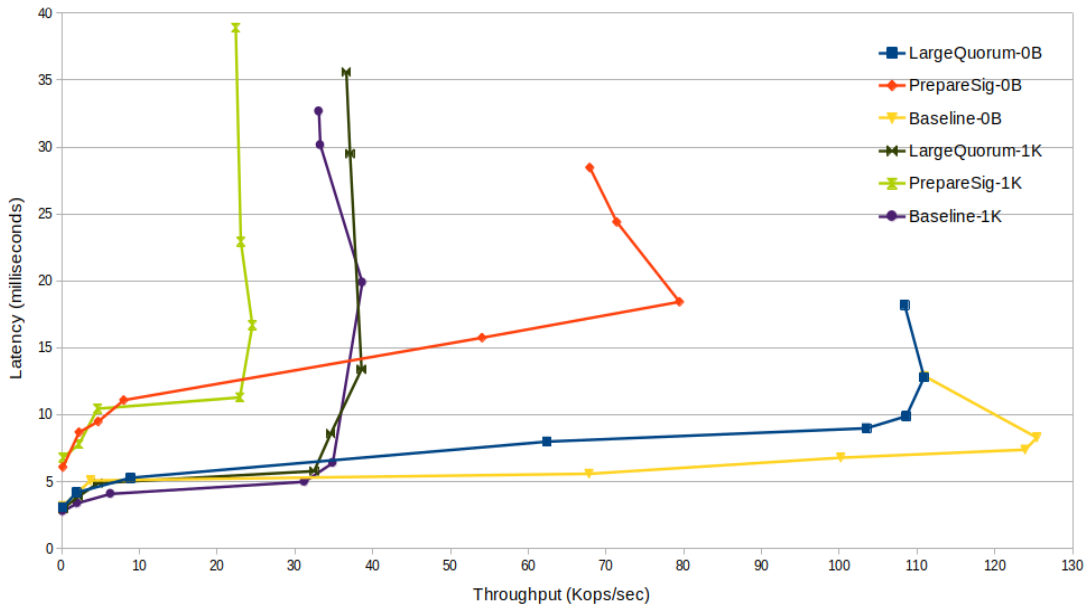


Figure 3.4: Throughput and latency for 0/0 and 1K/1K operations.

using two benchmarks. In the first benchmark, clients send requests of size 0 (empty), and replicas will also send back empty replies. Similarly, in the second benchmark, clients will send 1kB-sized requests, and replicas will send 1kB-sized replies. We call these benchmarks the 0/0 benchmark (introduced by Casto et al. [51] to measure the maximum possible throughput) and the 1K/1K benchmark, respectively.

For the 0/0 benchmark, increasing the quorum size for the replies causes clients to wait longer between requests, which increases the end-to-end latency and slightly decreases the throughput. As the payload and reply size increase to 1K, we see that the performance drops drastically due to the overhead of message transmission. This overhead is also the reason why the performance of the large quorum and small quorum are similar to the large payload since the time to transmit the message is greater than the time it takes for clients to wait for the extra reply.

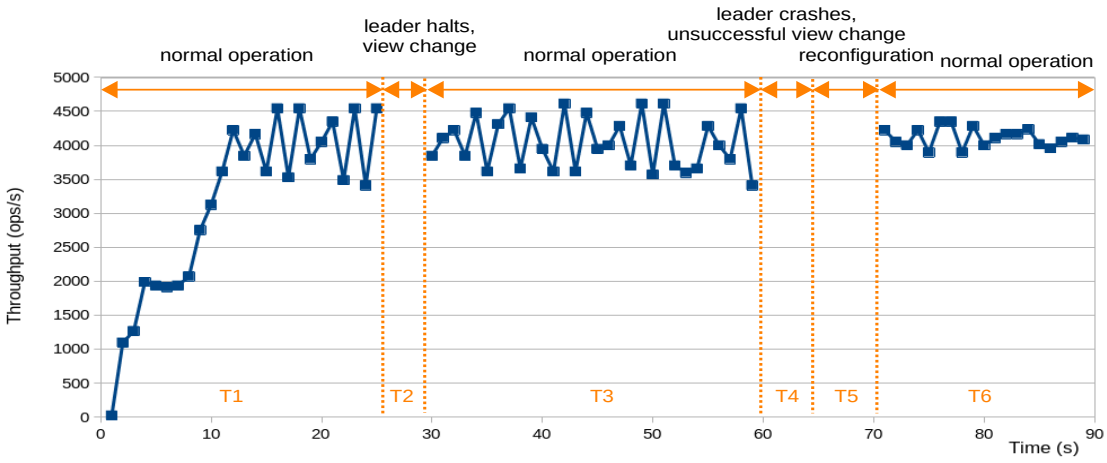


Figure 3.5: A system with $n = 4$ replicas going into reconfiguration after experiencing f_B and f_C faults.

For both benchmarks, adding signatures to the *prepare* phase greatly impacts the performance with respect to the other experiments. The reason is that in BFT-SMaRt, signatures in the commit are done speculatively; that is, when a replica receives a *prepare*, it spawns a separate thread to create and sign the *commit* before it even receives a quorum of the *prepare* message. This parallelism cannot be applied to the *prepare* message itself, as the replica has to wait to receive the *pre-prepare* before creating and signing the *prepare* message.

3.5.2 Reconfiguration

For the second experiment, we provisioned a cluster of four replicas and a total of 200 client processes distributed across four client machines. Here, we reduced the number of client machines and client processes because we are interested in reconfiguration latency but want the system saturated enough to visualize the drop in throughput. We configure the replicas so that each replica votes to remove another replica if it experiences faulty behavior from another

replica more than once. Figure 3.5 shows a timeline of the execution of the system with respect to its throughput.

In the time period **T1**, the replicas start up and establish connections with each other. Once done, the replicas are then able to accept connections from the clients and begin processing their requests. After 25 seconds, the Byzantine leader briefly halts, causing request timers on the other replicas to expire and the system to go into a view change. Time period **T2** shows the total time between the leader halting and operations resuming after a successful election, with 2 seconds of the time for the request timer to expire and another 2 seconds for the replicas to perform the view change. Because the Byzantine replica caused the view change to occur, all correct replicas will mark this replica.

During time period **T3**, the Byzantine replica behaves correctly again, then the system goes through a period of 30 seconds of normal operations, after which the new leader crashes. When the leader crashes, the communication channels that it has with the other replicas will be severed, and all will repeatedly try to reestablish the connection with the leader. The request timers will again expire after 2 seconds, causing another view change. However, the Byzantine replica now does not send any further messages because it suspects that the old leader has crashed after failing to reestablish the communication channel multiple times. This leads to an insufficient number of view-change messages and an unsuccessful view change in **T4**. Normally, this is the point where BFT-SMaRt without Phoenix would stop and perform these unsuccessful view changes, one after another.

Since the correct replicas only received view-change messages from each other, they each mark the other two (Byzantine and crash) replicas and, with the Byzantine replica having

two marks total, the correct replicas votes against the Byzantine replica. In **T5**, the CM receives the votes from the correct replicas, initiates a reconfiguration round, and subsequently removes the Byzantine replica. Finally, after a reconfiguration to replace the Byzantine replica, the new configuration has enough correct replicas to resume processing requests in time **T6**.

3.6 Related Work

Fault Detection. A failure detector can either be decoupled from the underlying distributed protocol to exist as an oracle that can be consulted, or coupled to allow protocol-specific actions to influence fault detection. Chandra et al. [55] introduced the concept of such oracles, but even the weakest failure detector cannot work in a fully asynchronous network due to FLP [90]. However, the assumption of partial synchrony can circumvent this result and allow such oracles to exist [56]. Malkhi and Reiter [151] defined a failure detector $\Diamond S$ that can detect *quiet* processes that do not participate in a protocol. Subsequent work by Doudou et al. [74] introduces a failure detector of the same class that can detect *mute* processes, the equivalent of *quiet* processes.

In BFT, failure detectors are typically coupled with the underlying consensus protocol. Haeberlen et al. [107] envisioned a fault detector that would allow replicas to determine if message omissions by a replica are malicious. Their approach requires replicas to frequently publish a signed digest of their log and to sign all messages it sends as well as to acknowledge all messages it receives. In addition to being inefficient, this approach to fault detection can be abused by malicious replicas that constantly issue challenges that require other replicas to

respond. Replicas in Aardvark [64] can detect subjective faults other than omissions, such as message flooding and invalid messages, and use these detections to blacklist faulty replicas. Regardless of whether fault detection is coupled or decoupled from the underlying consensus protocol, Phoenix can leverage the fault detector to make better reconfiguration choices.

Fault Recovery. Prior works on recovering faulty replicas in replicated systems fall into one of three categories: reconfiguration, proactive recovery, and reactive recovery. For CFT systems, protocols like Vertical Paxos [141] and Raft [168] reconfigure by having a special reconfiguration request committed by a quorum of the old active replica set as well as a quorum of the new active replica set. A faulty replica can also be recovered reactively by means of rebooting once it is detected [116], and there has been research looking into reducing the recovery time to increase availability [170] or by integrating diagnosis with recovery to support application-specific recovery actions [125]. All of these protocols work as long as there are only benign faults in the system, and would not work in the presence of Byzantine faults.

To reconfigure a replica set to remove a replica that could be Byzantine faulty, BFT-SMaRt [31] relies on a special process that sends a reconfiguration request to either add or remove a replica, which only needs to be committed by a quorum of the old active replica set. IA-CCF [191] allows a consortium of members in a permissioned ledger system to vote on referendums that update the current configuration. When there are enough votes, the new configuration can be carried out as part of a transaction that must be committed and executed. Earlier systems such as Rampart [180] and SecureRing [128] provide group membership protocols that can be used to reconfigure the replica set, but subsequent works have shown that these protocols are not ideal for the Byzantine fault model [50].

Various works have looked into proactively recovering replicas that could be Byzantine faulty [50, 179]. A process can rely on a timer at each replica to initiate a recovery process, which, in addition to rebooting, includes the replica discarding all the old keys used for encrypting communication that it had with the other replicas as well as the clients. Byzantine faulty replicas can also be recovered reactively with the help of a failure detector oracle [195].

Phoenix takes the approach of reactive reconfiguration to recover replicas because there are situations where faulty replicas cannot recover by a reboot [183], and also because it reduces downtime of the system. Proactive and reactive recovery requires rebooting of replicas and possibly re-establishing keys with clients and replicas, which can be expensive during runtime. However, this approach to recovery can complement reconfiguration. For example, a CM can decide to reintroduce replicas back into the system if they are still alive and have been rebooted and rekeyed.

3.7 Summary

Research in Permissioned BFT protocols has shown that there are various ways in which replicas can be faulty, and fault detection mechanisms have been proposed to detect these types of behaviors, but all of these mechanisms lead to an end result where the detected replica remains in the active replica set. Also, to be practical for real-world deployments where faults accumulate over time, these BFT protocols that perform state machine replication over a set of replicas need a way to replace faulty replicas to prevent the number of accumulated faults from surpassing a threshold; however, the reconfiguration mechanism in state-of-the-art BFT proto-

cols requires the administrator node to know the replicas that are faulty, which is not always possible. In this work, we introduced Phoenix, a novel reactive reconfiguration mechanism that integrates fault detection with reconfiguration to allow replicas to inform the administrator about suspected faults in order for the administrator to make better replacement choices. Our evaluation of Phoenix shows that the changes needed to facilitate reactive reconfiguration introduce only a small performance overhead.

Chapter 4

Phoenix Variants

4.1 Sync Phoenix: Stronger Recoverability Under Stronger Network Assumptions

4.1.1 Tolerating f_B and f_C with $n = 3f + 1$

Asynchronous BFT protocols cannot tolerate more than $\lfloor \frac{n-1}{3} \rfloor$ simultaneous faults out of $n = 3f_B + 1$ replicas [147]. A BFT system using Phoenix as described in Section 3 can reconfigure its active replica set as long as the number of Byzantine faults does not surpass f_B and the number of crash faults does not surpass f_C with a configuration of $n = 3f_B + f_C + 1$ replicas. It would be nice if Phoenix allows BFT systems to reconfigure under the same fault threshold but with a deployment of just $n = 3f_B + 1$ replicas. However, it is obvious that this is not possible because if there are f_C crashed replicas, a single Byzantine replica can prevent any reconfiguration by simply withholding messages during the reconfiguration phase. The CM can keep trying the reconfiguration request, but it will never succeed unless the Byzantine

replica participates. If we could somehow replace the f_C crashed replicas, the system will once again have enough replicas to drive consensus, even if the Byzantine replicas decide to continue withholding messages.

One approach to this problem could be to have the CM collect the latest system state (i.e., the longest decision log) and have replicas skip the synchronization phase and simply adopt this state. This would allow the remaining $n - f_B - f_C$ replicas to sync up without needing to reach consensus. The synchronization step in reconfiguration was needed so that any slow replicas can catch up if their log is missing the latest consensus decisions before moving on to the new configuration, like how replicas need to sync up before moving on to the next view in a view change. Having the CM be the one to collect each replica's state, determine the latest state by taking the longest matching log from $f + 1$ replicas, and then relaying it to the replicas in the new configuration achieves the same outcome.

4.1.1.1 Relying on the Client for State Transfer

This idea of relying on an entity for a state transfer between replicas is not new. Aublin et al. [19] describe a similar approach in their BFT protocol, Aliph, where the client is the one performing the task of collecting logs from the replicas, generating the history containing decisions from a set of longest matching logs, and then relaying this history. The insight behind Aliph is the composition of two classes of BFT protocols: optimistic BFT protocols that contain a speculative fast-path execution with optimal latency but require responses from all n replicas (i.e., no failures) [154] and permissioned BFT protocols like PBFT that can tolerate faults with higher latency.

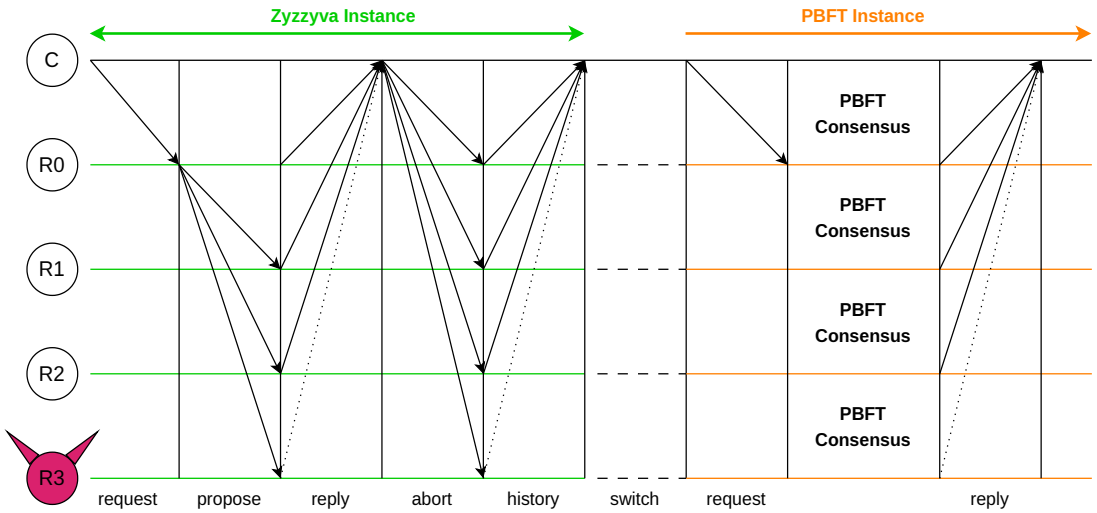


Figure 4.1: A client request in Aliph [19] fails to execute in the speculative fast path, and replicas abort to the slow path with the help of the client who collects and transfer the state between the two protocol instances.

Client requests are first sent to a set of replicas that execute an instance of an optimistic protocol, resulting in lower latency and higher throughput when the system is fault-free. If a client fails to get replies from all the replicas, as required to ensure safety in optimistic protocols, the request is then retried in the slower but more fault-tolerant instance of a permissioned BFT protocol. To safely transition between the two protocol instances, however, the replicas in the permissioned protocol must know where in the consensus decision log to start from and write decisions for new requests. This is done by having the client collect the logs from the replicas in the optimistic protocol, computing an *Abort History* that contains the longest matching log from at least $f+1$ replicas, and then relaying this history to the replicas in the permissioned protocol. Essentially, the client is responsible for the state transfer between the two protocol instances, thereby preserving the total ordering of requests.

Figure 4.1 shows how state transfer is done through the client in Aliph. Replicas initially start out executing an instance of their optimistic protocol, AZyzzyva (a version of Zyzzva [131] that contains just the speculative fast path), wherein requests are only successful if all n replicas execute the *propose* from the leader and reply to the client. One of the replicas, *R3*, is faulty and does not reply to the client, so there are not enough replies for the client to consider the request successful. After the client times out, it will inform the replicas to abort the current protocol instance and switch to a more fault-tolerant protocol by sending an *Abort* message. When a replica receives this *Abort* message, it will cease execution and send its log to the client. The client waits until it has received $n - f$ of these logs, and then computes a new log called an *Abort History* in which each entry in the log appears in $f + 1$ of the logs received from the replicas. A set of $f + 1$ matching values for each entry is sufficient in Aliph because digital signatures are used, so the entries in the *Abort History* are verifiable by replicas in the next protocol instance.

On the replica side, after sending the *Abort*, a replica will perform a deterministic switching operation that will change the current protocol instance it is executing over to an instance of *Backup*, their implementation of PBFT. When the client is finished constructing the *Abort History*, it will resend its request and include this history along with the request. Replicas will then iterate over the entries in the log, validating each one to ensure that the entries are valid ($f + 1$ signed messages for each entry) before adopting the log. Once the *Abort History* is processed, replicas will then execute the Byzantine commit protocol similar to PBFT and reply to the client. Here, *R*3 is still faulty and does not reply to the client, but because PBFT can tolerate f faults, the replies from the other replicas are enough for the client to consider the

request successful.

4.1.1.2 Relying on the CM for State Transfer

The work in Aliph has shown that it is possible for a single entity to mediate a state transfer between two sets of replicas, so long as the state constructed by the old replica set is verifiable by the new replica set. However, one problem that arises in relying on the client as done in Aliph is that the protocol is vulnerable to malicious clients. As shown above, the client has the power to force replicas to abort the current protocol instance, even if none of the replicas are faulty. Although in Aliph replicas eventually switch back to the optimistic protocol after some time, it does not prevent a client from continually forcing the replicas to abandon the fast path. At first glance, it seems straightforward that we can apply this approach in Phoenix by relying on the CM to construct a provable commit history, allowing us to trust the CM instead of the client, but we can run into problems if we are not careful.

Phoenix with *prepare* Signatures. To illustrate one such problem, consider Figure 4.2. Here we have a system of $n = 4$ replicas, with $f_B = 1$ and $f_C = 1$. A client sends its request, which successfully passes through consensus, but the only replicas that were able to gather enough *commit* messages are correct replica $R0$ (also the leader) and Byzantine replica $R2$. Still, this makes up the $f_B + 1$ replies that the client needs to consider the request successful and move on. After replying to the client, $R0$ crashes, and the system undergoes a view change. Suppose $R2$ correctly suspects that $R0$ has crashed and stops sending messages, the view change will not be able to complete as potential leaders won't be able to collect enough *view-change* messages.

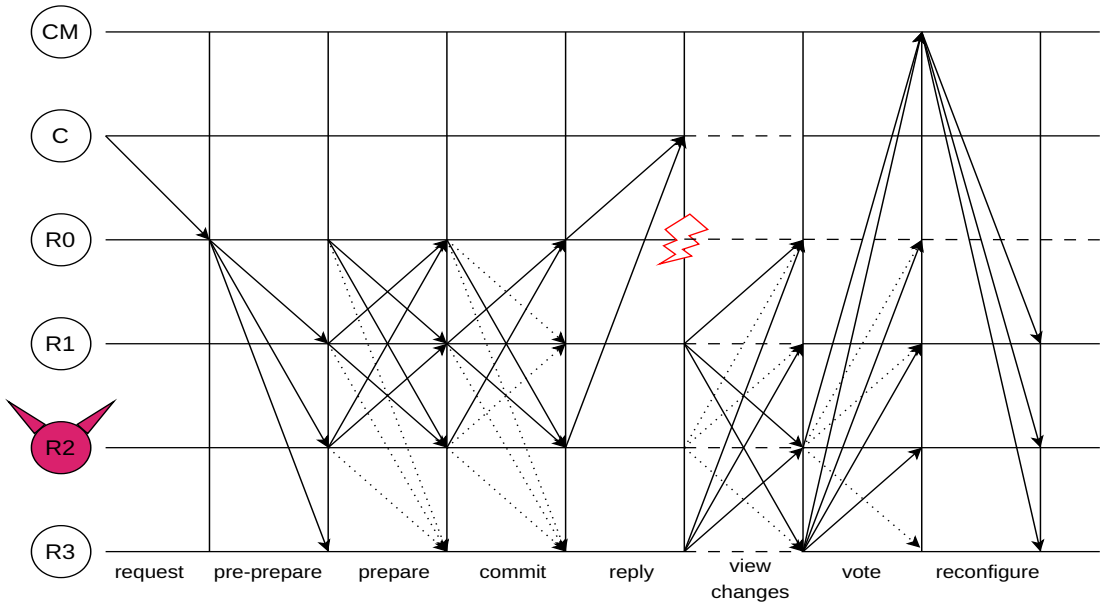


Figure 4.2: A client C receiving $f+1$ replies and considers a request successful, then the system exepriences f_B and f_C faults. Slow replica $R3$ sends its log in a vote, and Byzantine replica $R2$ truncates its log to match $R3$, causing the CM to construct a history without the decision for C's request.

Some time elapses before $R3$ decides that the system is stuck and enlists the CM for help in reconfiguration, sending its decision log in a *vote* message against $R0$, who was the initial cause of this issue. Since $R3$ is the slow replica, its log is missing the consensus decision containing the client's request, so its state is not the latest of all the replicas. $R2$ also happens to send its vote against $R0$ to the CM, but, being malicious, it truncates its log to remove the latest decision and match $R3$. The CM then uses the logs from $R2$ and $R3$ to construct a new log, which is then relayed to the replicas in a *reconfiguration* message.

When the replicas receive the *reconfiguration* message, they will verify and adopt this log and accept a new replica that replaces $R0$ into the active replica set. Recall that in Phoenix, we can have clients rely on a smaller quorum of f_B+1 reply by signing messages in

the *prepare* phase without violating safety in reconfiguration. This means that even though $R1$ has the *prepare* messages to prove that a decision was reached for the client's request, it will lose these messages when adopting the CM's log. Once replicas have been reconfigured, the position in the log where the client's request was decided is now empty and can be overwritten by a decision for a new request, thus violating safety.

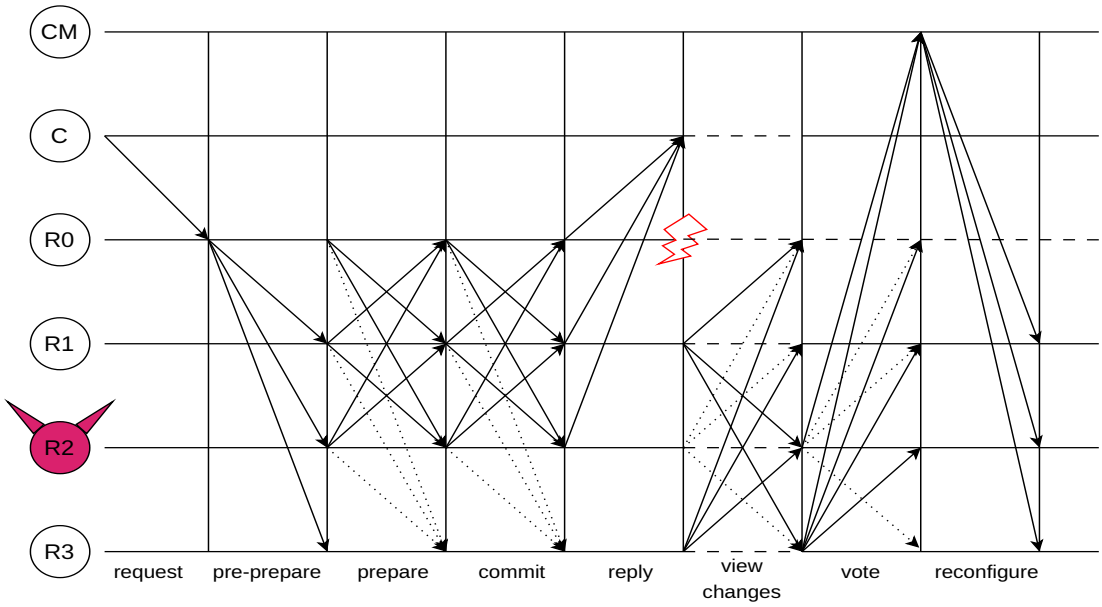


Figure 4.3: A client C receiving $n - f_B$ replies and considers a request successful, then the system exepriences f_B and f_C faults. Slow replica $R3$ sends its log in a vote, and Byzantine replica $R2$ truncates its log to match $R3$, causing the CM to construct a history without the decision for C's request.

Phoenix with Large Reply Quorum. We have shown that state transfer through the CM with Phoenix using *prepare* signatures could cause a safety violation. Now, we show that this is still the case if we configure Phoenix to use a larger quorum of replies. Consider a similar example shown in Figure 4.3, in which replicas $R0$, $R1$, and $R3$ all receive enough *commit* messages to

finish the consensus instance and reply to the client. Again, $R0$ crashes after replying, $R3$ is the slow replica that doesn't get any *prepare* or *commit* messages, and both $R2$ and $R3$ send their vote against $R0$ to the CM. The same situation occurs where the CM does not get to hear from a correct replica with the latest state, here $R1$, before it constructs a history log and issues a reconfiguration request, leading to $R1$ losing its commit certificate. Note that the CM in both cases cannot afford to wait for more votes in hopes of receiving a log from a correct replica with the latest state since it could end up waiting forever if the Byzantine replica does not send in its own vote. The only way for the CM to know that it has received at least one vote from a correct replica with the latest state is if all correct replicas can communicate with the CM in a timely manner during periods of reconfiguration, i.e., reconfiguration is strictly synchronous.

Given these observations, we now show how we can change our protocol Phoenix into a version that is able to tolerate f_B and f_C simultaneous faults with a deployment of just $n = 3f_B + 1$ replicas. Notably, these changes are:

- **State Transfer through the CM.** Similar to Aliph, replicas in the current configuration will relay their state so that replicas in the next configuration will all start with the same latest state, but using the CM as the mediator instead of a client.

- **Synchronous Reconfiguration.** The CM must be able to communicate with the correct replicas that have the latest state in a timely manner before constructing a new log that will be adopted by the replicas in the new configuration.

4.1.2 Changes to Phoenix

4.1.2.1 Voting

The first change is to require that replicas also include their decision log in their *vote* message when sending it to the replicas and the CM. A replica's decision log contains all the decided consensus instances, each with a proof [194]. The *vote* message from a replica will now be $\langle$VOTE, j, L_i, Proof $\rangle_{\sigma_i}$, where L_i is the decision log of the replica i.

When the CM or a replica receives a *vote* message, it checks that the decision log L_i has no gaps and that each decided value in the log has a valid proof. The checking of the decision log for gaps and valid consensus proofs is a standard check that is a part of the BFT-SMaRt view change algorithm. Once the contents have been validated, the CM will increment the vote count for the replica being voted against in the message, then records the decision log. When the vote count against a replica j surpasses $f_B + 1$ (i.e., at least one honest replica has voted against j), the CM then finalizes the voting round. To finalize a voting round, the CM sets a timer and sends a $\langle VOTE\_REQUEST, j, Proof \rangle_{\sigma_{CM}}$ to all replicas, requesting they send their vote. The CM waits until the timer has expired before processing the *vote* messages. In processing the *vote* messages, the CM determines the longest valid log that it has seen. It is sufficient to choose only a single log with the highest sequence number because the corresponding consensus decisions contain $n - f_B$ signed consensus messages. The CM then sends a $\langle$RECONFIGURE, $L_i, newConf \rangle_{\sigma_{CM}}$ to all the replicas, and sets a timer to wait for their responses. Here, L_i is the log from replica i that was chosen as the longest valid log.

4.1.2.2 Reconfiguration

When a replica receives the *reconfigure* message from the CM, it first checks that the message has a valid signature from the CM. Then, the replica will adopt the log L_k and change its configuration of active replicas to reflect that in the set S, after which it sends an acknowledgment to the CM to signal that it has successfully transitioned into the new configuration. The CM waits until the timer expires, by which time it will have received $n - f_B - f_C$ acknowledgments, before sending the *join* message to the replica that is joining the active replica set. At this point, all the replicas in the new configuration will initiate a view change to elect the leader for view $v + 1$, and then resume processing requests.

4.1.3 Sync Phoenix in Action

Now that we have discussed the changes to the Phoenix protocol that resulted in Sync Phoenix, let us see an example of the new protocol in action. Figure 4.4 shows an example of a system configured with $n = 3f_B + 1$ replicas, with $f_B = 1$ and $f_C = 1$. At some point in time, the current leader $R0$ crashes, causing the system to go through a view change to try and elect $R1$ as the new leader. Byzantine replica $R3$ (correctly) suspects that $R0$ has crashed and proceeds to withhold *view-change* messages, causing the system to go through a series of view changes without successfully electing a new leader. $R3$ could choose not to withhold the *view-change* message when it is its turn to get elected as the leader, but its goal is to halt the system, so subsequently withholding consensus messages after getting elected would achieve the same outcome as simply not getting elected at all. Similar to the situation in Figure 4.3, a correct replica $R2$ and the Byzantine replica $R3$ votes to remove $R0$ from the system after a series of

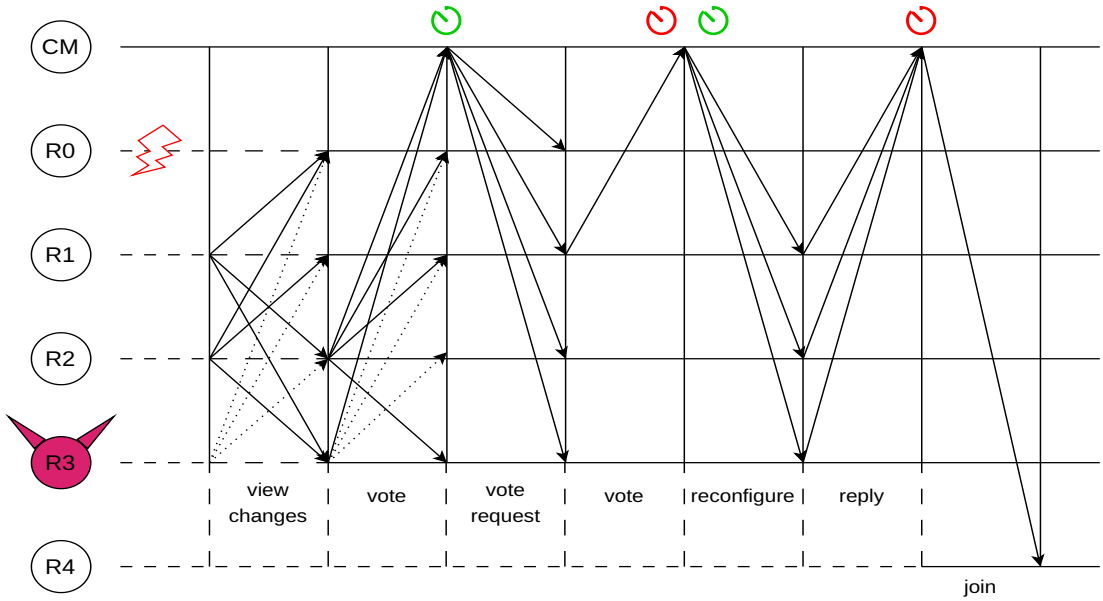


Figure 4.4: A system configured with $n = 3f_B + 1$ replicas ($f_B = 1$) experiencing f_B Byzantine faults and f_C crash faults simultaneously. Crashed replica $R0$ is voted out using the voting protocol in Sync Phoenix, and the CM transfers the state to replicas in the new configuration.

unsuccessful view changes. Once it receives $f_B + 1$ votes, the CM concludes the voting round, starts a vote timer, and sends a *vote-request* message to all the replicas. Replica $R1$, seeing that a voting round has concluded, sends in its own vote to the CM. With this vote from $R1$, the CM has received votes from all non-faulty replicas in the current active replica set, including the ones with the most up-to-date state.

Using the logs *vote* in the *vote* messages, the CM determines which log is the longest valid log and includes that in a *reconfigure* message, which is sent to all the replicas. It also sets another timer to wait for the acknowledgments from the replicas. As described in the previous section, the replicas will adopt the log contained in the *reconfigure* message from the CM before reconfiguring its view of the active replica set and reply to the CM. Even if Byzantine replica

	n	CQ	VQ	RQ	Reconfiguration
BFT-SMaRt [31]	$3f_B+1$	$n-f_B$	$n-f_B$	f_B+1	$n-f_B$
Phoenix	$3f_B+f_C+1$	$n-f_B$	$n-f_B$	$n-f_B$	$n-f_B-f_C$
Sync Phoenix	$3f_B+1$	$n-f_B$	$n-f_B$	$n-f_B$	f_B+1

Table 4.1: The differences in sizes of the Commit Quorum (CQ), View-change Quorum (VQ), Reply Quorum (RQ), and Reconfiguration Quorum (RQ) for BFT-SMaRt, Phoenix, and Sync Phoenix.

$R3$ does not reply to the CM, replies from $R1$ and $R2$ make up the $n-f_B-f_C$ replies needed for the CM to consider the reconfiguration successful. The CM then relays the same log sent in the *reconfigure* message in a *join* message that is sent to the new replica $R4$ that is joining the active replica set. $R4$ will also adopt this log, after which it will connect with the other replicas and then start to participate in the current configuration.

4.1.4 Quorum Sizes

Table 4.1 shows the differences in the quorum sizes and how they influence reconfiguration. BFT-SMaRt changes to a new configuration by committing it as a request, therefore, reconfiguration requires a quorum the size of the commit quorum, CQ. Committing configurations as requests means that the clients can keep the reply quorum, RQ, at f_B+1, and the system guarantees that finished requests will persist across reconfigurations. A system using Phoenix requires a higher number of replicas provisioned ($n=3f_B+f_C+1$), but it can tolerate f_C crash faults in addition to f_B Byzantine faults. Since new configurations cannot be committed if there are f_B and f_C faulty replicas, reconfiguration is done through the view change with a VQ of $n-f_B-f_C$. Clients must wait for $n-f_B$ replies, and maintaining CQ at $n-f_B$ prevents a Byzantine replica from successfully overwriting committed values. Sync Phoenix is able to

tolerate f_B and f_C faulty replicas whilst keeping n at $3f_B + 1$ by relying on synchrony during reconfiguration to maintain safety. Similar to BFT-SMaRt, CQ and VQ are kept at $n - f_B$, and reconfiguration only requires $f_B + 1$ in the worst case, but clients must also wait for $n - f_B$ replies.

4.1.5 Summary

In this chapter, we presented Sync Phoenix, a version of Phoenix that is able to tolerate f_B Byzantine faults and f_C crash faults with a deployment of just $n = 3f_B + 1$ replicas if the underlying network is synchronous. Sync Phoenix is able to preserve the most up-to-date state of the system by relying on the CM as the mediator for state transfer between the old and new configuration. This approach is similar to another BFT protocol, Aliph, which relies on the client to perform the state transfer, but state transfer through the CM in Sync Phoenix is not vulnerable to Byzantine clients that can force replicas to abort the execution of consensus instances. We show that the changes to Phoenix to construct Sync Phoenix are minimal, only affecting the voting and reconfiguration phases, and the phases in the Byzantine commit protocol remain the same.

4.2 Crash Fault Tolerant Reactive Reconfiguration

The novelty behind reactive reconfiguration in Phoenix presented in Chapter 3 is that by aggregating localized fault detection information at the replicas to the CM, we enable it to detect and replace replicas that it wouldn't have otherwise suspected. In this chapter, we show that this approach is not limited to BFT systems and that reactive reconfiguration can also be used in crash fault tolerant (CFT) systems.

4.2.1 Reconfiguration in Raft and Zookeeper

Unlike BFT systems that can tolerate Byzantine replicas, CFT systems can tolerate replicas that may only fail by stopping. Because of this, the underlying consensus protocol is less complex, involving fewer rounds and messages exchanged for each consensus instance, as there is no risk of equivocation. However, like BFT systems, CFT systems also need a way to dynamically change their active replica set to replace faults if the service is to remain highly available.

Two examples of CFT systems that enable reconfiguration are Raft [168] and Zookeeper [117], both widely used in practice [24, 199, 175, 208]. Figure 4.5 shows the reconfiguration process in Raft, which uses what is called *joint consensus*. A client (presumably an administrator) issues a reconfiguration request to the current leader $R0$ to replace replica $R1$ with $R3$. $R0$ appends to its log an entry consisting of the old and new configuration and sends an *AppendEntries* command to replicas in both the old and new configuration to append it to their own log. Joint consensus requires that a quorum of replies from both configurations is needed

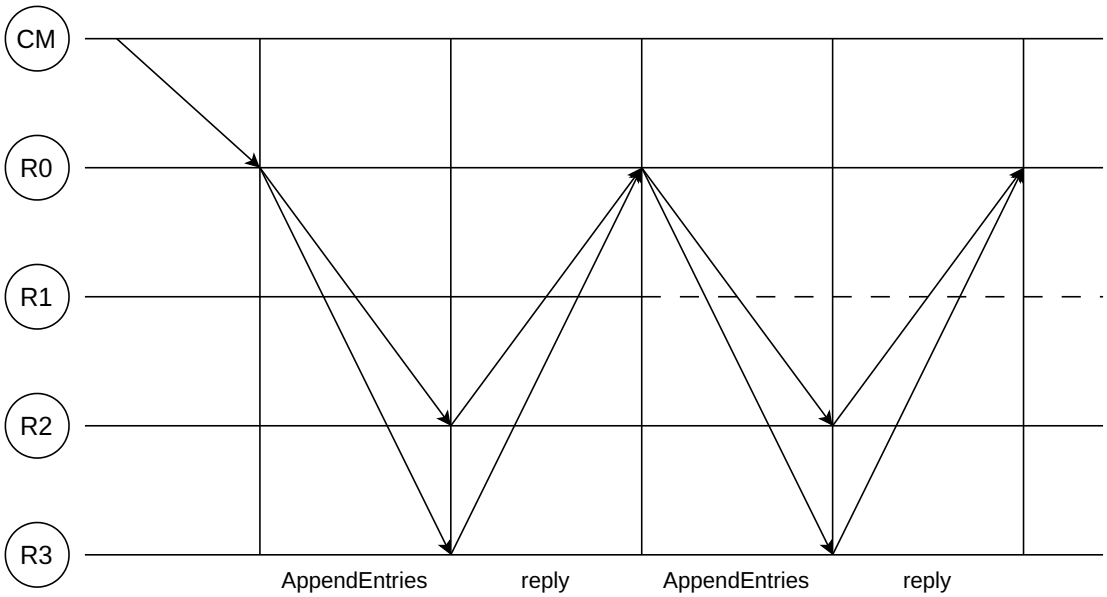


Figure 4.5: Reconfiguration in Raft using joint consensus.

in order for the $R0$ to continue; here, the replies from replicas $R2$ and $R3$ satisfy this requirement. After $R0$ receives the replies, it will append an entry containing only replicas in the new configuration to its log, and once again sends an *AppendEntries* command, but this time only to replicas in the new configuration. Once a quorum of replies is received, the reconfiguration is complete, and the system can continue to process client requests.

4.2.2 Reducing False Positives and False Negatives using Reactive Reconfiguration

Similar to BFT-SMaRt, reconfiguration in Raft and Zookeeper relies on a reconfiguration request from a special client (CM) to replace a faulty replica. The CM in these two systems must also have a way to detect faults to know when to issue the reconfiguration requests. In

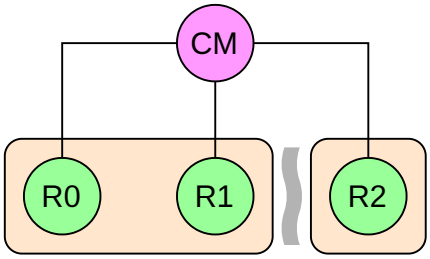


Figure 4.6: CM uses fault detection using heartbeats, but cannot detect that replica $R2$ is partitioned away from the rest of the replicas.

Zookeeper, for example, using heartbeat intervals has been shown to be the best fault detection strategy [178]. However, the use of heartbeats as a means for fault detection can lead to false positives and false negatives [198].

An example showing the limitation of fault detection using heartbeats is shown in figure 4.6. Here, replica $R2$ is partitioned away from the other two replicas, $R0$ and $R1$, and cannot participate in consensus. But, the connection between $R2$ and the CM is not affected, so when the CM issues its heartbeat messages to all the replicas, $R2$ is able to respond normally, signally to the CM that it is still alive. $R2$ will remain useless as long as the partition persists, which could be hours or even days, depending on the underlying issue [23]. Ideally, the CM would replace $R2$ with another replica that can communicate with $R0$ and $R1$ so that any subsequent failures will not cause the system to become unavailable. If reactive reconfiguration was instead used, replicas $R0$ and $R1$ would notice that $R2$ is not responding to their own heartbeat messages and send their vote to the CM, indicating that $R2$ is faulty, allowing the CM to replace $R2$.

Figure 4.7 shows another example of the limitation of fault detection using heartbeats. In this example, replica $R0$ is partitioned away from the CM, but it is still connected to the other replicas. This means that $R0$ is still participating in consensus as usual, but it is not

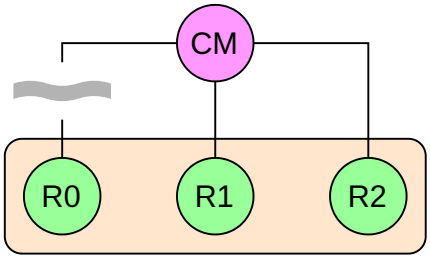


Figure 4.7: CM uses fault detection using heartbeats, but detects that replica $R0$ is down and replaces it.

able to respond to the CM's heartbeat messages. To the CM, $R0$ seems faulty and needs to be replaced, so eventually, it would issue a reconfiguration request to replace $R0$. Replacing $R0$, in this case, would cause an unnecessary impact on the system's performance, and the best way to avoid it would be for the CM to simply wait for the partition to heal and let the replicas continue processing requests as normal. With reactive reconfiguration, the CM would not initiate a reconfiguration unless it receives *vote* messages from $R1$ and $R2$ indicating the $R0$ is faulty, which would not happen in this case.

Chapter 5

Fault Detection in Blockchain Systems: Detecting Eclipse Attacks

In this chapter, we will present our algorithm for detecting eclipse attacks in Proof-of-Work (PoW) blockchain systems. We will also present an evaluation of this algorithm by running it against all historical blocks in the Ethereum blockchain. At the time of writing, PoW was the dominant consensus algorithm used by major blockchains valued at hundreds of billions of dollars, such as Ethereum [213] and Bitcoin [163], so eclipse attacks on miners and users on these networks have the potential to cause large financial losses. This concern has led to various works on addressing eclipse attacks [153, 216, 174, 8, 61, 27]. However, most of these works focus on the source of the attack, which is the connections that the victim machine has with the attacker and does not leverage information that is provided by the blockchain consensus mechanism.

Each block that is produced by the network contains a rich amount of information that

makes the block unique and helps other participants in the network easily validate the block. One such piece of information is the block's difficulty value, θ, which we will monitor in our mechanism for detecting eclipse attacks. The fluctuations in θ depend on the network's hashing power, Q, with a higher Q resulting in a higher θ due to blocks being mined at a faster rate, and vice versa. Earlier in Section 2.2.2, we mentioned that if the attacker controls only a minority percentage hashing power q out of Q, a PoW consensus algorithm prevents the attacker from producing blocks at a rate that could cause inconsistencies in the main chain (i.e., the canonical chain that the majority of the network agrees on). For this work, we choose $q = 30\%$ as an upper bound to make our exposition more concrete. We argue that this percentage is a reasonable upper bound since the biggest Ethereum mining pool at the time of writing has a hashing power of 25% [172], but we make this parameter configurable in our implementation. An attacker that possesses a large q could withhold their hash power from the rest of the network to mine blocks in secret, but this would lower the difficulty of future blocks, making it attractive for new miners to join, undermining this strategy for the attacker. During normal operation, θ is typically around $13 \cdot Q$, targeting an expected block arrival time of 13 seconds.

To clarify one subtlety, the block arrival time in Ethereum accounts for the hash power spent on mining the next primary block as well as the *uncle blocks*, which are valid blocks that arrived later than the previous primary block. Based on Ethereum's Difficulty Adjustment Algorithm (DAA) formula [41, 98], primary blocks that have uncle blocks may arrive within 18 seconds, instead of nine seconds to avoid lowering the difficulty value. However, splitting the hash power in half between uncle blocks and primary blocks to get twice the time does not result in a strategic advantage. Therefore, we assume the attacker spends its hash power generating

blocks on a single malicious fork and produces no uncle blocks.

During an eclipse attack, the attacker must generate blocks every nine seconds to avoid a difficulty drop [98]. Initially, this is difficult for the attacker, but each time the attacker fails to produce a block on time, the DAA lowers the difficulty by a maximum of five percent. For an attacker with $30\% \cdot Q$ hash power, θ will eventually approach a point where the attacker can expect to mine blocks fast enough to maintain the difficulty level indefinitely. We define the difficulty at which the attacker can maintain a block arrival rate of nine seconds as $\theta_{min} = 9 \cdot (q \cdot Q)$.

Since θ_{min} is the boundary at which the attacker has complete control over the chain, applications may wish to set a lower difficulty drop as a cutoff point. The lower the $diffSym_{min}$, the more confirmations are necessary to have a high assurance that the most recent blocks were not mined by the attacker. Setting a more conservative bound on the drop will require fewer confirmations, but this comes at the price of potential false positives: blocks that trigger our algorithm while the system is not under attack. We evaluate this tradeoff in detail in Section 5.3. Once the difficulty approaches θ_{min}, however, no number of confirmations can eliminate the possibility of an attack.

Ethereum's DAA adjusts the difficulty value from block to block, so we must monitor the difficulty drops over time to prevent attackers from incrementally dropping the difficulty value. Given that the difficulty level adjusts naturally as miners join and leave the network, we need to establish a stable difficulty level for a given time span and adjust this level over time. Beginning at the genesis block, the chain is split into fixed-length checkpoint ranges of N_c blocks. We set the current difficulty level as the median of the blocks in the previous (complete)

checkpoint range. N_c should be large enough to make it impossible for an attacker to maintain a $diffSym$ higher than θ_{min} after N_c blocks but small enough to avoid false positives.

5.1 Known Mitigation Strategies

The work of Heilman et al. [110] in which the eclipse attack was realized in Bitcoin also proposed a strategy to help mitigate the attack by diversifying peer connections. The approach works by limiting the number of connections that originate from the same IP address, so an attacker would need to control a higher number of machines to form more connections to the victim. A subsequent work by Marcus et al. [153] proposed an improvement to this strategy, in which a node could enforce a mapping between IP addresses and ECDSA keys, preventing the attack from forming more than one malicious connection per machine that it controls. Xu et al. [216] proposed a strategy that involves analyzing the peer connections and filtering the incoming and outgoing packets to these connections to detect commonalities that suggest they could originate from the same attacker. Since a successful eclipse attack relies on establishing a large amount of connections over a long period of time, the work of Prünster et al. [174] suggests to periodically refresh the peer routing table so that the attacker would lose some of the established connections to the victim node. Although these strategies could be effective in some cases, fault detection and mitigation are done on the network layer and do not take advantage of the information contained in the blocks produced by the consensus layer. Alangot et al. [8] presented a solution that does not rely on the network layer but instead uses the block timestamps to detect eclipse attacks.

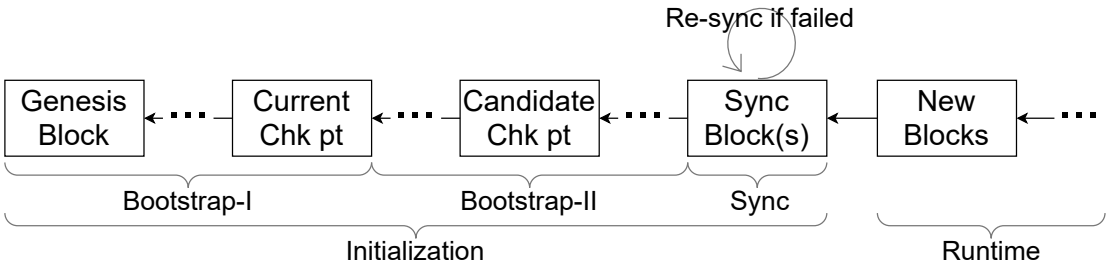


Figure 5.1: Difficulty Monitor initialization

Ekiden [61] addresses eclipse attacks using a Proof-of-Publication (PoP) protocol, which requires a nonce to be published within a transaction to the chain and the victim to receive a block containing that nonce within some time window. Having periodic blocks that contain a nonce prevents the attacker from withholding blocks from the victim for prolonged periods of time or accumulating blocks to perform a selfish mining attack. The solution presented by Bentov et al. [27] relies on block arrival time and also requires that new blocks on the network are received within some time threshold. Relying on just the block arrival time is insufficient because an attacker could take advantage of periods when the network difficulty drops to produce these blocks at a rate fast enough to meet the expected time. Moreover, periods of difficulty fluctuations in blockchain networks are common, leading to a high variance in arrival times, so choosing the right time threshold is difficult.

5.2 Monitoring Block Difficulty

5.2.1 Monitor Phases

Our difficulty monitor has three initialization phases, shown in Figure 5.1, which ensure the legitimacy of the existing blocks.

Bootstrap-I Phase The monitor loads and processes all blocks starting from the *genesis block* to the latest *checkpoint block*. Blocks associated with well-known false positives (such as hard forks), are classified as *confirmed benign blocks* and included when determining the difficulty of the checkpoint range they appear in.

Bootstrap-II Phase Next, the monitor loads all blocks published after the checkpoint block. Difficulty changes will be checked with the same process used at runtime but without restrictions on the elapsed time.

Sync Phase Finally, the monitor synchronizes with the blockchain by publishing a sync message, similar to the PoP used by Ekiden. Unlike Ekiden, we only require a sync message at initialization. This prevents attackers from feeding a pre-mined malicious fork to the monitor. Thus, it ensures the freshness of both the *Sync block* and following blocks. Sync messages contain a nonce generated by the monitor, and a block containing the sync message with the matching nonce must be received within the expected block time, which is around 13 seconds in Ethereum. If the sync message fails to be published within the restricted time limit, the host will need to perform a re-sync by publishing a new nonce until a sync block is accepted.

5.2.2 Runtime Monitoring

New blocks received by the monitor are first validated as normal. Then, the monitor determines the nominal block time based on the block timestamps. If the difference between the trusted elapsed time and the nominal block time is greater than 15 seconds, the block is rejected to prevent the attacker from either lowering the difficulty in the primary chain by manipulating timestamps or maintaining the difficulty in the malicious chain with pre-dated timestamps. Temporary forks in the chain require each branch to be monitored simultaneously. Our documentation discusses further details [223].

Once the new block has passed the initial validation, the monitor compares θ_{new} presented in the new block with the θ_{min} calculated based on the formulas shown below.

$$\theta_{\text{ref},i} = \text{Median}([\theta_{(i-1)\times N_c}, \cdots \theta_{i\times N_c}]) \tag{5.1}$$

$$\theta_{\text{min},i} = \theta_{\text{ref},i} \times (1-p) \tag{5.2}$$

The monitor first gets the median of difficulty values among the blocks in the previous checkpoint window, which has been checked and are finalized. Finalized blocks are blocks with a large number of successors and negligible probability of being reorganized. By using the median as a reference value, we minimize false positives and false negatives caused by outliers. Next, the monitor uses θ_{ref} to calculate the θ_{min}, where p is the percentage drop allowed (set by the application). The new block is accepted if θ_{new} is higher than θ_{min}.

The attacker may choose to provide no blocks at all, preventing the monitor from determining how much the difficulty value has dropped. To address this issue, the monitor will

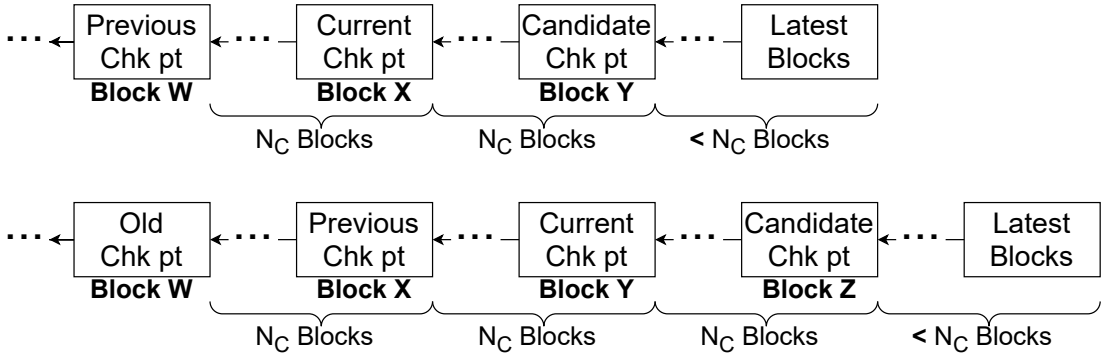


Figure 5.2: Checkpoint Update

periodically estimate the current difficulty value, θ_{est}, by using the trusted elapsed time, and compares θ_{est} with θ_{min}. Additionally, the maximum allowable block time in Ethereum's DAA (around 900 seconds) prevents the difficulty value from dropping more than 5% in each block. This value also serves as the maximum wait time ($T_{\Delta Max}$), and blocks arriving later than this time will trigger a shutdown.

5.2.3 Checkpoint Update

The monitor will automatically update the checkpoint every N_c blocks. As shown in Figure 5.2, the candidate checkpoint is the block that has less than N_c blocks ahead and is too new to be considered finalized. Once there are N_c blocks found ahead of the candidate checkpoint, the candidate will become the new checkpoint, and the N_cth block ahead of the new checkpoint will become the next candidate checkpoint.

Test Case Index	θ_{min}	N_c	Num of False Detections	False Negative Rate
1	$50\% \cdot \theta$	1270	2	$< 2^{-128}$
2	$60\% \cdot \theta$	880	2	$< 2^{-128}$
3	$70\% \cdot \theta$	620	3	$< 2^{-128}$
4	$80\% \cdot \theta$	430	5	$< 2^{-128}$
5	$80\% \cdot \theta$	610	7	$< 2^{-256}$
6	$90\% \cdot \theta$	275	17	$< 2^{-128}$
7	$90\% \cdot \theta$	420	23	$< 2^{-256}$

Table 5.1: Evaluating the difficulty detection algorithm under various checkpoints sizes against the entire Ethereum database.

5.3 Evaluation

We implemented our methodology on top of Geth [82], the official Go implementation of the Ethereum client since it is the most mature and widely used client. To evaluate the false positive rate under different parameters, we ran the monitor's algorithm against 10,900,373 historical blocks in Ethereum. We released our custom Geth database connector, experiment code, and more technical details at [224, 223].

5.3.1 False Negative Rate

The false negative rate, as shown in Table 5.1, represents the possibility of an attacker maintaining the difficulty value at the level higher than θ_{min} within N_c blocks. It is calculated based on the exponential distribution [27] established with the attacker's hash power and network difficulty [223]. All combinations of N_c and θ_{min} that we have examined result in extremely low false negative rates, equivalent to brute-forcing a 128 or 256-bit key.

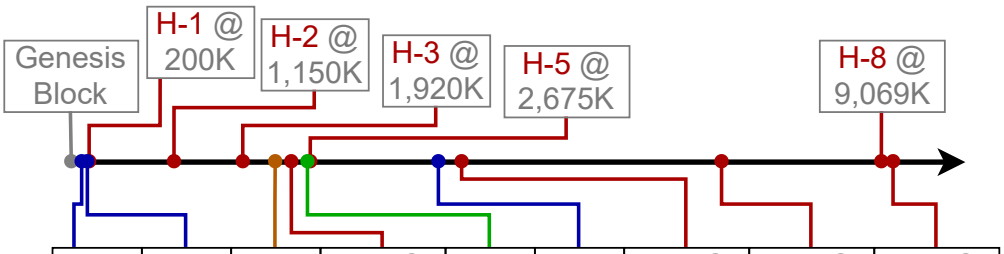
5.3.2 Difficulty Monitoring and Checkpoint Sizes

Table 5.1 also shows false positives that triggered our difficulty monitoring algorithm under various $\theta_{\min}$ and N_c. For all configurations, the algorithm resulted in multiple false detections. We have investigated all of these false positives and found that, as shown in Figure 5.3, Most of these events correspond to hard forks. During hard forks, network hash power fluctuates significantly due to miners upgrading their nodes. There are also instances where the detections were caused by DoS attacks [211] addressed with software upgrades. In both cases, the Ethereum client would need to be shut down and replaced by one built with the latest protocol anyway, so the monitor's behavior is not as concerning.

When $\theta_{\min}$ is set to a high value, such as $90\%\theta$, the monitor becomes more sensitive to smaller events, such as bugfixes and API updates. In the 7th test case, there is one false positive caused by a 10% difficulty drop that we could not tie protocol-related events but could be due to miners leaving for Ethereum Classic, which was increasing in value at the time.

When $\theta_{\min}$ is set to a higher value, the monitor needs a smaller checkpoint size (i.e., N_c) to confirm that the recent N_c blocks have a negligible chance to be mined by the attacker. As N_c increases, so does the number of false positives since the checkpoint captures a larger time interval surrounding hard forks. Thus there is a tradeoff between N_c and $\theta_{\min}$. The application may set a larger $\theta_{\min}$ or N_c to make malicious forks more difficult but may incur more false detections since difficulty drops accumulate over a longer period.

Therefore, setting a reasonable $\theta_{\min}$ is a key to avoid false positives, and then configuring a reasonable N_c to reach the desired false negative rate level. For instance, with $\theta_{\min}$ set



	B-1	B-2	A-1	H-4 @ 2,463K	U-1	B-3	H-6 @ 4,370K	H-7 @ 7,280K	H-9 @ 9,200K
False Pos. By Test Case:	1 to 7	7	3 to 7	6, 7	6, 7	7	1 to 7	5, 6, 7	7

Table Legends:

H: Hard Fork | **B: Bugfix** | **A: Attack** | **U: Update**

Figure 5.3: Timeline of major events that cause false detections. The hardforks are [205], [212], [42], [119], [120], [84], [121], [122], and [123], respectively. The bugs are [196], [197], and [85], respectively. The attack is [211]. The major API update is [83].

to 80%θ, all those false positives can be avoided if the upgrade is timed correctly; a lower N_c with a 70%θ or lower will also have the same sensitivity to difficulty drops.

5.3.3 Block Times

Currently, the average block time in Ethereum is around 13 seconds. To find a proper value for the $T_{\Delta\text{Max}}$, which is used to prevent attackers from remaining silent forever, we record the top 100 block times from historical blocks in Ethereum. Our results show that while the majority of the blocks do arrive within the expected time, all of the top 100 blocks took over 200s to arrive, with 91 blocks taking less than 300 seconds, 7 blocks less than 400 seconds, 1 block less than 500 seconds, and one outlier that took 13013 seconds. Upon closer inspection, almost all of these blocks were mined very close to the Byzantium hardfork [84]. The reason

that the outlier block took almost four days to be mined is because of a consensus issue (which is B-1 in Figure 5.3) in Geth that caused miners to have to switch to alternative clients[196]. Given these results, choosing the maximum time given in Ethereum's DAA (around 900s) should give no false positives during normal operation; for applications that are very time-sensitive, a reasonable $T_{\Delta\text{Max}}$ would be 400 seconds.

5.4 Summary

In this chapter, we addressed the problem of eclipse attacks in blockchain systems which, unlike a similar attack in non-blockchain systems, could cause financial losses to the users of the network and presented a mechanism for detecting these attacks in PoW blockchains. Our mechanism works by monitoring the difficulty value of the blocks produced by the network, and determine if the difficulty of new blocks remains within their expected values based on the network's hashing power. We evaluate our algorithm by running it against all historical blocks in the mainnet of the Ethereum blockchain, showing that the false negative rate is extremely low for various assumptions of the attacker's hashing power and that the algorithm did trigger false positives, but all of these are the result of significant events in the Ethereum network that would require any entity relying on our monitor to stop and upgrade their software.

Chapter 6

Fault Detection in Blockchain Systems: Detecting Execution Forks

In Section 2.2.3, we discussed the scalability (or lack thereof) of blockchain systems and why scaling solutions that move data or computation off-chain, such as state channels, are needed for these systems to handle a large number of users. Being one of the most popular scaling solutions, vulnerabilities to attacks such as the Execution Fork Attack places its users and their funds at risk. Current solutions to Execution Fork Attacks are not sufficient in that they require an external committee of third-party members to attest transactions from payment channel users before they are submitted, or delegate the responsibility of detecting the attacks and submitting the most up-to-date state to a custodian that is required to always be available. Solutions that introduce an external committee not only introduce overhead but also inherit a weaker availability guarantee from the Byzantine fault tolerant protocols that they use to send messages between committee members. As for solutions that involve a custodian to monitor

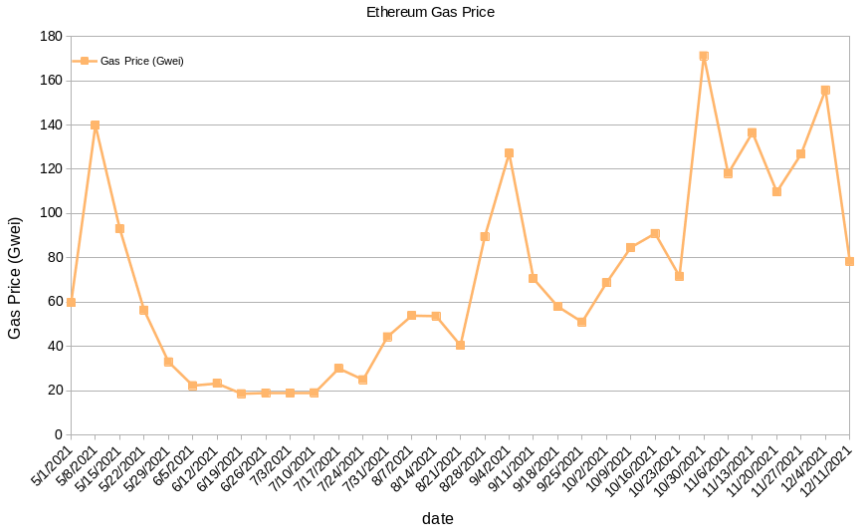


Figure 6.1: Gas price to send a transaction on Ethereum over time

the payment channel, fault detection is not enough to ensure that the honest party does not lose money, which can happen when the network becomes congested and the cost to send transactions outweighs the money that they would lose from an attack.

Figure 6.1 shows the ”gas price” for Ethereum, which dictates the price of sending a transaction on chain [2]. As the network becomes congested, this gas price will increase, making the cost of sending a transaction more expensive. Suppose that round i above was not the latest transaction between Alice and Bob, and that Bob sent Alice a large sum of money between rounds i and the most recent round $i + k$. If the on-chain state of the channel in the smart contract reflects transaction i, and Bob initiates a withdrawal of his on-chain balance, Bob could cheat Alice out of the money he sent her in the last k rounds. In order to not lose money, Alice needs to refute Bob's withdrawal by sending a *dispute* transaction that contains the off-chain state for round $i + k$ before the challenge period Δ_T ends. However, if the cost to

send transactions is too high due to a congested network, Alice might elect to send a transaction with a low fee, which might not make it within Δ_T. After a successful withdrawal by Bob, Alice would not be able to withdraw her balance reflected by round $i + k$, even if her *dispute* transaction makes it to the contract because there are no longer sufficient funds in Bob's on-chain balance to pay Alice.

In this chapter, we present a simple and novel redesign of smart contracts for payment channels that would help mitigate execution fork attacks. We realize that the only way to compensate Alice for transaction fees that could exceed the remainder of Bob's balance is by having a security deposit. With a deposit separate from Bob's transaction balance, Bob would forfeit his deposit to Alice if she successfully disputed his on-chain balance update and withdrawal attempt. We present a construction of payment channels that provide compensations for successful disputes when network fees are high and provide a proof-of-concept implementation of the on-chain contract in Solidity.

6.1 Known Mitigation Strategies

To mitigate execution fork attacks, payment channels must ensure that valid dispute transactions make it to the contract before the challenge period ends. One way to do this is to set a large Δ_T, which would give the network more time to decongest and for Alice's low-fee transaction to be included in a block. While a large Δ_T would work for brief periods of congestion, Ethereum and Bitcoin have both experienced prolonged network congestion that lasted for months on end. Setting a Δ_T that encompasses even these long periods of congestion

is not practical because it would cause funds to be locked up for too long, even if the party making the withdrawal is honest.

Another challenge in mitigating execution forks is ensuring that honest parties are online to be able to dispute; if Alice is offline during the challenge period, she cannot dispute even if she is willing to pay a high transaction fee. Previous works proposed appointing a custodian that could help dispute Bob's withdrawal on Alice's behalf if she is offline [3, 156, 4]. When Alice receives the payment from Bob for round $i+k$, she can sign and relay the message to the custodian, who constantly monitors the blockchain for messages to the contract. The custodian can thus send a transaction on Alice's behalf if it sees that Bob is attempting an execution fork attack. These solutions still do not address the issue of high fees during congestion. If Bob's off-chain balance is zero after round $i+k$, a successful dispute still causes the custodian to lose fees for the transaction. The only way to reimburse Alice (or the custodian) for the dispute is to have a separate balance for Bob that is untouched by their off-chain transactions, and this deposit can then be used to cover the cost of the dispute transaction if it is successful. Our work realizes this missing part in the solutions to execution fork attacks and aims to provide a better construction of payment channel contracts to ensure fairness for both parties in the presence of high fees.

The authors of Brick [20] proposed a solution that prevents one party from unilaterally closing the channel using a stale state by introducing a committee of third party members (called *Wardens*) that needs to attest the closing state before it can be accepted by the channel contract. However, this solution only works for permissioned blockchain systems, and the inclusion of a group of wardens introduces overhead both off-chain and on-chain (identities of wardens are

stored on the contract). FastPay [25] extends on Brick by introducing payment contracts that also require a committee but supports payments between more than two members. Collins et al. [66] introduced a similar system, called Astro, wherein the committee members are a set of replicas that executes Byzantine reliable broadcast [45] to propagate transactions between users and store them locally instead of relying on the blockchain. These solutions in which a committee needs to use a Byzantine fault tolerant protocol to attest for off-chain payments inherit a weaker availability guarantee since at most f out of a committee of size $n = 3f + 1$ are allowed to be Byzantine.

6.2 Building Robust Payment Channels

6.2.1 Constracts with Deposits and Reactive Withdrawal Period

Construction. Figure 6.2 shows our construction of a bi-directional payment channel. When Alice and Bob want to form a payment channel, they first need to reach an agreement on the conditions of the channel: initial balances and time to wait before they can withdraw in the number of blocks. After agreeing on the terms, they exchange signatures on this message and send a transaction to create this contract. The function **constructor** will initialize the contract based on the agreed-upon terms. A deadline is set to infinity because it only starts when one party signals the intention to withdraw. The contract also records the party that closes the channel and is punished.

```
1   Contract PaymentChannel
2       on constructor(bals, deps, withdrawwait)
3       balances := bals
4       Δ_T := withdrawwait
5         deadline := ∞
6         _round := 0
7         deposit := ∅
8         closingparty := ∅
9         punished := false
10
11      on update(bals, round) by sender
12        require _round > round
13        require sender in balances
14        for i := 0 to bals.len
15          if bals[i] != balances[i] and
16          closingparty != ∅ and
17          punished := false
18            balances[¬closingparty] += deposit
19            punished := true
20          balances[i] = bals[i]
21        _round := round
22
23      on withdraw(bals) by sender
24        require currentblock > deadline
25        amount := balances[sender]
26        if sender == closingparty and !punished
27          amount += deposit
28        send amount to sender
29
30      on withdrawExit(bals, round, deposit) by sender
31        if closingparty == ∅ then
32          update(bals, round)
33          deposit := deposit
34          deadline := currentblock + Δ_T
35          closingparty := sender
36          emit (withdrawExit, sender)
```

Figure 6.2: Smart contract for bi-directional payment channels.

Balance Update. The payments that are sent by Alice and Bob are only reflected in their off-chain state. To sync up the on-chain state with the off-chain state, either party can call the **update** function, which will assign the on-chain balances with the off-chain balances sent in the message if all the signatures are valid. It is required that the update must be using a state that is more recent than that in the smart contract, and the contract enforces this by checking the round number.

Withdraw. To close the channel and withdraw money, either party can call the **withdrawExit** function. They will also include their off-chain state of the channel, which will be used to update the on-chain state. The smart contract will signal an event to notify the other party that the channel is being closed to withdraw money. At this point, the network could be under heavy network congestion, and the closing party could be malicious and try to withdraw from the channel using an old off-chain state that benefits them. The closing party is required to include a *deposit*, which will be held by the smart contract and returned back to the closing party if there is no dispute or used to compensate the other party if there is a successful dispute. If the other party has no objection before the deadline, the contract will be closed by returning the remaining balance to both parties and the withdraw deposit to the closing party.

In section 6.3.1, we discuss how we utilize a new functionality provided by Ethereum smart contracts, which allows the contract to view the block's base fee. To withdraw the balance, the requesting party not only needs to pay the transaction fee to submit the request, but also needs to make a temporary deposit to the contract that is the same amount as the fee to send a dispute transaction. By using the same amount of transaction fee of the dispute request, we can have an accurate estimate about how much it will cost to submit a dispute request under current condition.

Dispute. If the contract is about to be closed with a older state, the other party may submit a dispute to update the latest state. In the traditional smart contract, under a congestion situation, the disputing party has a risk of losing money. With the dispute deposit, the disputing party no longer needs to worry about losing the money because once the dispute is approved by

the smart contract, the withdraw deposit will be paid to the disputing party in order to cover the cost of submitting a dispute to the smart contract. The **update** function also serves as a dispute resolution mechanism. If one of the parties tries to close the channel and the other party submits a state that is more recent, then the closing party will be assumed to be malicious, and their deposit amount will be forfeited.

Withdrawal Deposit. Our design withholds the deposit in the minimum amount and in the shortest time. By requesting the withdraw deposit to have the same amount as the cost to submit the request, we can avoid both cases of undercharge (if the cost for submitting a dispute is actually higher) and overcharge (if the cost for submitting a dispute is actually lower). Compared to making an additional deposit at the construction time, making the withdraw deposit with the closing request only requires the closing party (instead of both parties) to make the deposit, and the deposit is only withheld from when the closing request is received to when the contract is closed.

6.2.2 Extending to Payment Channel Networks

Payment channels can be used to construct payment channel networks [79, 78, 145], which link together bi-directional channels and allow parties to send multi-hop transactions between each other. Two parties that want to transact do not even need to have a direct channel with each other provided that there's a route between them and that each hop has sufficient funds to make the payment. These payment networks also suffer from the same vulnerability to execution fork attacks as a single payment channel. A malicious party that attempts the attack

could cause a cascading effect if successful. Suppose there's a network of channels between Alice, Bob, and Trudy. If Trudy cheats Bob out of his funds, Bob might not have enough left on his balance to be able to pay Alice. In this case, it is crucial that Bob's dispute transaction makes it to the contract, which can be done by enlisting a custodian. A custodian watching the blockchain for Trudy's malicious withdrawal can submit a dispute, preventing Bob from having insufficient funds to pay Alice while being compensated for the dispute transaction from Trudy's deposit.

6.3 Evaluation

6.3.1 Solidity Implementation

As a proof-of-concept, we implemented our payment channel contract in Solidity [69], and tested the contract on Ethereum's Rinkeby testnet. Much of the contract translates directly from the pseudocode in Figure 6.2; however, we highlight several important differences. Firstly, we assign the *closingparty* based on the party that sent the transaction to *withdrawExit*. Initially, this variable is set to the zero address "address(0)", and it is used to check two conditions:

1. the dispute process can only be initiated once, and

2. the closing party does not try to dispute its own transaction and claim the deposit.

Secondly, Solidity contracts have access to the block number in which the transaction was executed, allowing it to set a deadline that is Δ_T number of blocks from the current block. Finally,

```
function update(parameters...) public checkSignatures(_signatures, _balances, _nonce) {
    require(nonce > _nonce)
    for (uint i = 0; i < _balances.length; i++) {
        if ((balances[users[i]] != _balances[i]) &&
            (closingparty != address(0)) &&
            (closingparty != msg.sender) &&
            (punished == false)) {
                balances[msg.sender] += deposit;
                punished = true;
            }
        balances[users[i]] = _balances[i];
    }
}

function withdrawExit(parameters...) public payable onlyUser checkSignatures(_signatures, _balances, _nonce) {
    bool validDeposit = checkDeposit(msg.value);
    if(validDeposit) {
        update(_balances, _signatures, _nonce);
        deposit = msg.value;
        deadline = block.number + withdrawWait;
        closingparty = msg.sender;
        emit WithdrawExit(msg.sender, _nonce, msg.value);
    }
}

function checkDeposit(uint _deposit) public view returns (bool) {
    if(_deposit >= block.baseFee * disputeGasUnits) {
        return true;
    }
    return false;
}

function withdraw() public onlyUser {
    require(block.number >= deadline);
    uint amount = balances[msg.sender];
    balances[msg.sender] = 0;
    if(msg.sender == closingparty && !punished) {
        amount += deposit;
    }
    (bool sent, ) = msg.sender.call{value: amount}("");
    require(sent);
    emit Withdraw(msg.sender, amount);
}
```

Figure 6.3: Solidity contract based on pseudocode in Figure 6.2.

the latest London update in Ethereum allows the contract to also have access to the block's base fee, allowing it to enforce a deposit amount from the closing party that is exactly equal to the cost of sending a dispute transaction at the current point in time. An honest party looking to dispute this withdrawal thus does not have to worry about the cost of the transaction if they were to immediately send the dispute. This feature is unique to Ethereum, and it discourages a

party looking to perform an execution fork attack to do so when the network fees are high. We released the code for our Solidity payment channel, which can be accessed online [201].

function	gas cost	median cost $	max cost $
update	56217	17.19	28.62
dispute	76781	23.48	39.09
withdrawExit	134379	41.09	68.41
withdraw	34749	10.62	17.69

Table 6.1: Cost of calling each function on Ethereum's mainnet

6.3.2 Cost of Transactions

Given our implementation of the contract in Ethereum, we evaluate the cost of executing each of the functions. Table 6.1 shows the functions in the contract and their corresponding gas cost and cost in USD. To get the median and maximum dollar amount, we looked at the base fee for the most recent 1000 blocks on the mainnet and used the values in these blocks along with the gas cost. As expected, the cost to send these transactions on the mainnet would be very high given the current state of the network, even if we consider the median fees. This shows that any payment channel currently deployed on the mainnet, where the difference between the on-chain and off-chain balances benefits a party within the price to dispute, is susceptible to a malicious execution fork attack.

6.4 Summary

Payment channels are one of the leading scalability solutions for Ethereum, allowing users to transact simply by signing messages and avoiding high transaction fees and latencies.

However, these channels are vulnerable to execution fork attacks that allow an attacker to capitalize on the difference between the on-chain and off-chain state of the channel and cheat the other party out of their funds. In this paper, we show that the current solutions to detect execution fork attacks are not enough, and they only solve the problem of ensuring a successful dispute without any regard for the high cost of sending these dispute transactions during periods of congestion. We presented a novel construction of payment channels that solves this problem in execution fork attacks.

Chapter 7

Fault Detection in Blockchain Systems:

On-chain Fault Detection

In the previous chapter, we explored how to enhance smart contracts in state channel networks to detect `Execution Fork` attacks and improve one of the major scaling solutions that exist in blockchains today. Smart contracts are trustless, self-executing code that exists as part of the data in the blockchain itself, so they inherit the high availability and integrity guarantees provided by the underlying blockchain. Furthermore, because the contract code is public, anyone can verify its state transitions without relying on a trusted third party. These contracts are written in a high-level language (e.g., Solidity) that contains a rich set of features, so they can be used in a wide range of applications and fault detection is not limited to simply detecting malicious state updates.

Recent research has shown that smart contracts can be used to encode a membership management mechanism on-chain [184, 155], wherein a group of servers will run an application

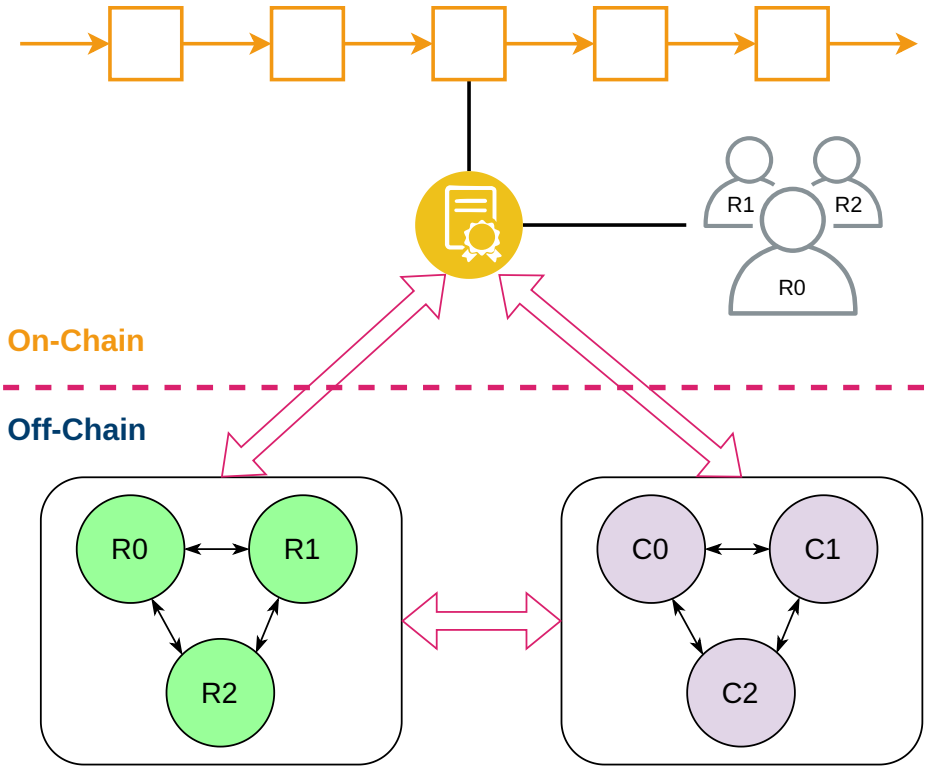


Figure 7.1: A smart contract managing membership for a group of replicas.

off-chain, and the on-chain contract will manage their identities and record their actions. Figure 7.1 shows an example of such a deployment in which a smart contract encodes the membership of three replicas that provides a service to clients. The replicas in this application process client requests and send replies to clients directly, but they will also record the outputs for these replies on chain by sending messages to the smart contract. Since clients and replicas communicate directly, the service is able to guarantee low latency for clients. Additionally, because the sequence of operations is persisted on-chain by the replicas, slow or newly-joined clients can monitor the contract to audit all of the operations that the replicas have performed,

even if all of the replicas fail.

These works, however, lack a way to detect faults in the replicas themselves and rely on a trusted third party to reconfigure the set of active replicas. For example, in Chainlink [37], reconfiguration commands to the smart contract can only be sent by an administrator who is presumed to have knowledge of the faulty replicas, which, as shown in our work in Phoenix, is difficult since the administrator does not participate directly in normal operations. In order to fully utilize the trustlessness of smart contracts in these hybrid on-chain and off-chain applications, we need a membership management mechanism that is done entirely within the smart contracts themselves without the intervention of an administrator. However, this would require a smart contract to not only record outputs from the replicas but also authenticate digitally signed messages from the replicas, allowing the contract to validate and aggregate localized fault detection information relayed by the replicas.

In this chapter, we explore the possibility of enhancing smart contracts in decentralized applications with automated fault detection mechanisms. First, we present smart contracts for membership management in applications that provide services to clients using a group of off-chain replicas. We encode in these contracts methods to authenticate digitally-signed messages from replicas, and provide functions within the contracts to automatically detect and replace faulty replicas using these signed messages. We use these contracts to build a decentralized, highly-available publish/subscribe (pub/sub) system called Decentagram, the first pub/sub system that supports on-chain authentication and on-chain subscriber notification. Additionally, we show that Decentagram can be used to remove the need for trust in a configuration manager that is used in Phoenix (Chapter 3), and also to further improve our robust payment channels

(Chapter 6) by preventing known malicious parties from initiating a channel-closing operation. Then, we present a smart contract that can be used as a heartbeat failure detector. For decentralized applications in which two (or more) untrusted parties encode service level agreements in smart contracts, it is difficult for one party to compel the other to meet their end of the agreement. Our smart contracts encode an incentivized dispute-response mechanism that can be used by one party to issue a challenge and punish the other party if they do not respond within a predefined time period (in the number of blocks). In essence, this amounts to encoding a heartbeat mechanism inside the smart contract, which will treat the non-responding party as faulty if they fail to respond in time. Using this contract, we build a platform for blockchain-based serverless computing, called AlacriTEE, that provides resource accounting and reputation management.

7.1 Detecting Faults Through Equivocation, Leaked Credentials, and Voting.

In this section, we present Decentagram, a decentralized framework for data dissemination using the publish/subscribe messaging model. Decentagram uses blockchain smart contracts to authenticate events that will be published using digital signatures, with these signatures being verified on-chain. This approach permits any host with valid credentials to publish verified updates, increasing decentralization and availability of the system as a whole by simplifying compensation and incentivization. Decentagram also supports on-chain subscribers where third-party contracts receive events immediately: within the same transaction as the published event. The same event will also be delivered to off-chain subscribing applications through

an off-chain event broker. We provide an open-source implementation of Decentagram, and evaluate the gas cost of its on-chain components and the end-to-end latency of its off-chain component.

The goal of publish/subscribe (pub/sub) systems is the dissemination of information from *publishers* to interested *subscribers* quickly and efficiently. Several production systems have been developed (e.g., [132, 101, 175, 108, 11]), and pub/sub systems have been applied in a variety of contexts, from financial applications [150, 57] to health monitoring [89], as well as real-time vehicle detection [133, 134]. To declare interest in a category of published events, subscribers register with a *broker*. Publishers send their events to the broker, possibly including metadata indicating relevant categories, and the broker notifies registered subscribers of the new event.

Some recent systems [176, 226] integrate pub/sub systems with blockchains to provide Byzantine fault tolerance and auditability of published events. However, these systems rely on Byzantine fault tolerant (BFT) protocols for consensus among off-chain replicas before publishing events to the blockchain. Unfortunately, the trust assumptions between blockchain protocols and these off-chain BFT protocols are mis-matched. In addition to the blockchain protocol's trust assumptions (e.g., attackers control less than 51% of staked ETH [193]), subscribers must also trust an additional entity to manage membership in the BFT protocol. The membership management mechanism (MMM) is implicitly trusted to authenticate replicas, select the system parameter f that specifies the maximum number of Byzantine faults to tolerate, and maintain the required number of member replicas needed. Note that even if the MMM is distributed among the replicas in some fault-tolerant way, the subscriber must still trust that the

parameter f is sufficient to prevent malicious events from being published, and that the non-faulty hosts—as viewed by the protocol—are trustworthy from the subscriber's perspective.

Decentagram is the first pub/sub system that supports decentralization of *data oracle*, *publishers*, *brokers*, and *subscribers*, and is also the first blockchain-based pub/sub system that can deliver events on- and off-chain simultaneously. Decentagram's on-chain smart contract framework allows replicas to authenticate with digital signatures. Authenticating digital signatures on-chain provides Decentagram with three significant advantages over previous systems:

- First, Decentagram enables *permissionless publishing*, allowing any host to run a data oracle.
- Second, Decentagram supports *incentivized event availability*, which compensates publishers for timely event publication and enables nullification of any (rational) benefit gained by delayed or suppressed events.
- Third, Decentagram replaces trust in third-parties (such as a quorum of replicas) with much weaker trust assumptions: other than blockchain assumptions, subscribers only need to trust in the authenticity of the digital signatures.

Decentagram also supports on-chain subscribers: third-party contracts that receive published events. Propagating events to third-party contracts in previous systems [226, 215, 176] requires off-chain subscribers to monitor blocks for new events and submit a transaction with the event to the third-party contract. This Monitor-and-React (M&R) approach introduces a delay between when an event is published and when the client's smart contract can react, making it challenging to process events consistently across on-chain and off-chain subscribers.

During periods of network congestion, the time between on-chain and off-chain reactions could increase if the relayed events are not included in the next block.

Enabling timely authenticated on-chain notifications is especially useful for applications with non-deterministic smart contracts that require data from an external party to make decisions [9]. For example, in Decentralized trading markets [80] in which clients monitor an on-chain marketplace contract for the latest price updates to make bids, an on-chain client can react immediately (within the same block) and make time-sensitive bids much faster than M&R clients (at least 1 block away). As another example, applications that utilize smart contracts to manage the membership of a group of off-chain compute nodes [37] also benefit from on-chain notifications. In this case, if any of the compute nodes are compromised, its credentials need to be revoked, and immediate notification of the on-chain membership contract will prevent the compromised node from any further participation in on-chain activities.

To the best of our knowledge, Decentagram is the first pub/sub system that delivers on-chain events directly. This functionality uses gas limits on cross-contract calls to safely execute callbacks to third-party contracts, preventing broken or malicious contracts from launching gas-exhaustion attacks [18]. Furthermore, using Decentagram's incentivized event availability, the increased cost of publish transactions incurred by these callbacks is paid with subscriber fees, not by data oracles.

In summary, this paper makes the following contributions.

- We present the design and implementation of Decentagram, the first decentralized pub/-sub framework with on-chain digital signature verification.

- Decentagram is resilient to Byzantine failures and is highly available, allowing any host to run as a data oracle as long as it can authenticate itself to the on-chain broker contract.
- Decentagram's smart contracts support on-chain subscribers for clients, and provides atomic on-chain event notifications for subscribers to instantly react to events.
- An evaluation of the on-chain smart contracts shows that the gas cost to publish and subscribe is reasonable, and that the throughput of our off-chain components is more than sufficient to handle events in new blocks.

7.1.1 Categories of Pub/Sub Systems

Systems	Fault Tolerance	Fault Threshold	Auditability	On-chain Notification	Publishing Incentive
Linda [46], SIENA [47]	✗	✗	✗	✗	✗
ISIS [34, 33], Kafka [132], RabbitMQ [175]	Crash	$\lceil \frac{n-1}{2} \rceil$	✗	✗	✗
HyperPubSub [226]	Crash	$\lceil \frac{n-1}{2} \rceil$	✓	✗	✗
Trinity [176]	Byzantine	$\lceil \frac{n-1}{3} \rceil$	✓	✗	✗
Chios [75]	Byzantine	$\lceil \frac{n-1}{3} \rceil$	✗	✗	✗
Chainlink [37]	Byzantine	$\lceil \frac{n-1}{3} \rceil$	✓	✗†	✓
Decentagram	Byzantine	Blockchain	✓	✓	✓

Table 7.1: Representative Pub/sub systems. †Most Chainlink clients access oracle data off-chain, but some on-chain processing for special *aggregator* contracts is possible.

Table 7.1 shows the features of representative Pub/sub systems and how they compare to Decentagram.

Early work in pub/sub systems such as Linda [46] and SIENA [47] described how, in loosely-coupled distributed systems, events can be generated and consumed by a set of processes, but did not consider the possibility of machine failures. To provide availability in the presence of failures, data can be replicated across a set of servers, as is done in ISIS [33, 126] and modern industrial pub/sub systems like Kafka [132] and RabbitMQ [175], where a subset

of the servers can fail by crashing. One common feature between these crash fault tolerant Pub-/sub systems is the ability to provide causal delivery of events to subscribers. Causal ordering of events ensures that events sent by multiple publishers to a subscriber are delivered in the same order that they were sent. Though sufficient for some applications, a stronger reliability guarantee such as publication total order [75], which ensures that all subscribers receive events in the same order, may be required for many applications (e.g., stock market data).

Some recent systems make use of blockchain technology to eliminate centralization of the event broker, tolerate Byzantine failures, and provide auditability for publications [176, 226]. HyperPubSub [226] keeps a record for each pub/sub operation on the blockchain, but the system does not fully tolerate Byzantine faults because the broker is implemented using Kafka which, as mentioned earlier, can only provide fault tolerance against crashes. Trinity [176] solves the issue of Byzantine brokers by using a Byzantine fault tolerant consensus protocol, and a Byzantine quorum of brokers (i.e., more than two thirds) need to agree on an operation before a transaction is sent to the blockchain to record the operation. While Byzantine fault tolerant consensus improves resilience against malicious brokers, they are limited by the fault threshold $f = \lfloor \frac{n-1}{3} \rfloor$, where n is the number of brokers. This means that if there are n brokers, at least $n - f$ brokers must be operational for the system to function, and the system will be unavailable if the number of accumulated faults surpasses f. Chainlink [37] describes a system with an on-chain oracle contract that can validate reports generated by a set of off-chain oracles. A designated off-chain leader oracle collects attestations to form a report and submits it to the oracle contract, which then determines the oracles that contributed to the report and compensates them. Each client can then monitor the oracle contracts for events that are emit-

ted, and then send transactions to update their contract. The system, however, requires that the oracle contract belongs to an administrator who can not only add or remove oracles, but also set the compensation amount for the oracles. Clients can also interact with the oracles directly, essentially becoming their own on-chain broker, but this requires paying oracles per request.

In Decentagram, the broker is implemented as a smart contract, so the system inherits the blockchain's availability guarantees. For public blockchains, these are typically much stronger than what is possible to achieve with traditional BFT protocols. In Ethereum, there are currently over 974K validators and 24M ether staked in the Ethereum mainnet [26], meaning that at least one-third [193] of them must be compromised to prevent the blockchain from proceeding. The system remains available as long as the blockchain is available and at least one oracle can provide authenticated publication. BFT protocols have been demonstrated to scale only up to the low hundreds of nodes [29], implying blockchain availability guarantees are at least three orders of magnitude higher. Thus Decentagram remains available under significantly more faults than systems such as Chainlink, Trinity, and Chios. Decentagram also improves on monitor-and-check approaches such as Trinity [176] by delivering events to subscribing contracts immediately. In Section 7.1.5 we compare the average, minimum, and maximum latency experienced by an application based on Decentagram and one based on the M&R approach.

7.1.2 Decentagram: Design

Figure 7.2 shows the architecture of the Decentagram framework, which consists of off-chain data oracles (DO_O)[1] and on-chain publisher (Pub_C), on-chain brokers, off-chain bro-

[1] In the following, we subscript off-chain Decentagram components with *O* for *off-chain*, and on-chain components with *C*, for *contract*.

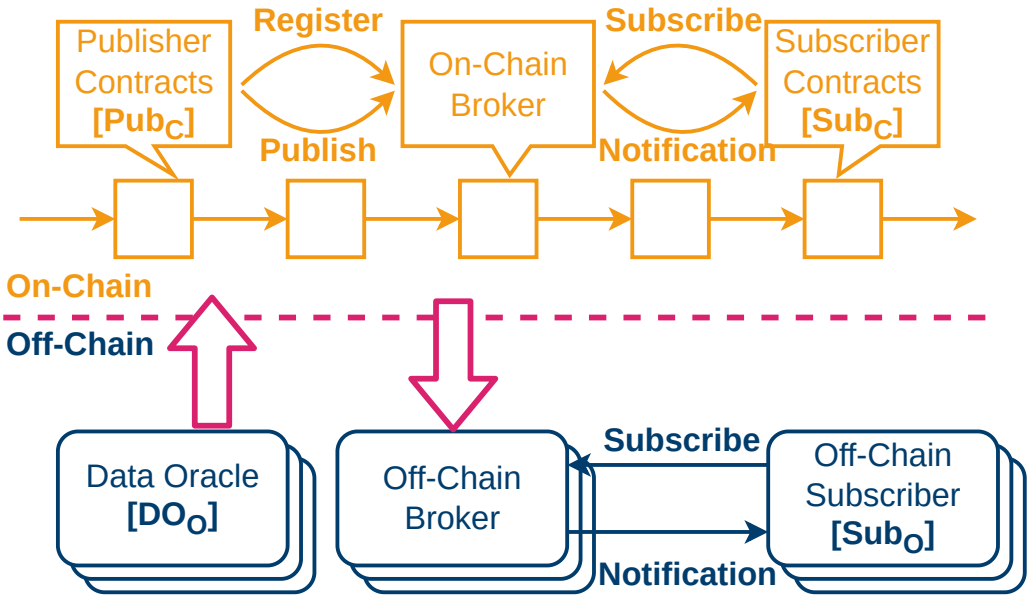


Figure 7.2: Overview of Decentagram

kers, on-chain subscribers (Sub_C), and off-chain subscribers (Sub_O). DO_O is the source of event data, so it knows what and how to collect data needed by the application. DO_O publishes a new event by making transactions, containing the event data, to the blockchain. The on-chain broker is implemented by a set of smart contracts that incentivize the publication of events and disseminate these events to Sub_C. Sub_O uses the off-chain broker to subscribe to events and receive notifications. Upon receiving a new event, the on-chain broker emits an EVM event, which includes the DO_O's event data in the current block. The off-chain broker then filters the EVM events in each block for those from the on-chain broker contract address.

Decentagram is made up of four types of smart contracts: PubSubService (PS_C), EventManager (EM_C), Publisher (Pub_C), and Subscriber (Sub_C). With respect to traditional Pub/sub systems, the PS_C and EM_C together form the on-chain broker, which indirectly con-

nects publishers and subscribers. PS_C is the first contract deployed on the blockchain, after which all publishers and subscribers can use it by referencing its address. For each new event type that it registers, PS_C will deploy a new EM_C that handles the stream of events for that type. Each Pub_C is the entry point for off-chain data oracles (DO_O). When the DO_O sends transactions with data to its designated Pub_C, the Pub_C is responsible for authenticating the DO_O's data. If DO_O is a trusted source or is providing digitally signed data from a trusted source, the Pub_C can verify the digital signature on the data directly. Otherwise, when DO_O is operated by an untrusted host, Pub_C authenticates the DO_O's TEE to verify it is running known and trusted code, ensuring its content is trustworthy.

Among these components, DO_O, Pub_C, Sub_C, and Sub_O are application dependent, and implemented by application developers. We assume these known and trusted implementations correctly use Decentagram libraries, so all authenticated TEEs contain correctly implemented Decentagram components.

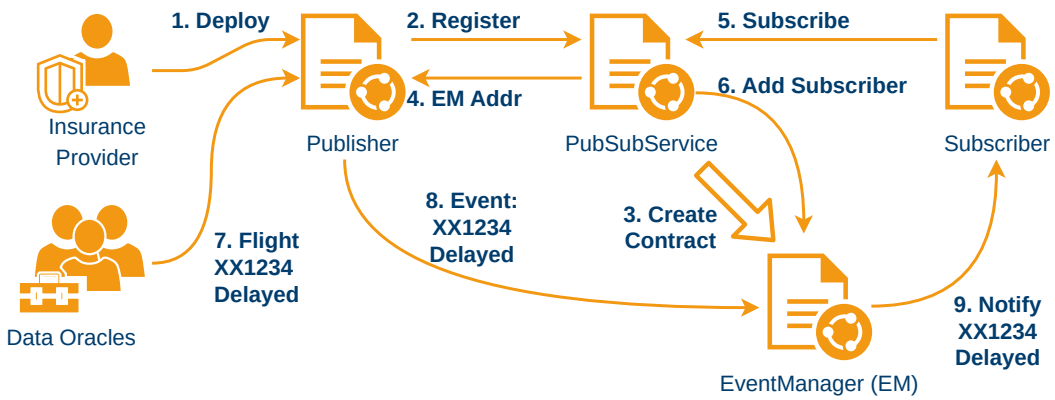


Figure 7.3: Registering a new Publisher and adding a Subscriber.

On-chain Pub/sub. Figure 7.3 illustrates the interactions between these contracts for the travel insurance example. At deployment, (1) the Pub_C registers itself with the PS_C (2), and receives the address of the new EM_C created by PS_C (4). To subscribe to these on-chain events, a Sub_C calls PS_C (5), specifying the address of the Pub_C it wishes to receive events from and a deposit to compensate the transaction cost of their event notifications. PS_C calls the relevant EM_C (6) to add the new Sub_C's address and callback function to its list of on-chain subscribers. For each update received and *verified* by the Pub_C (7), the Pub_C calls into the EM_C (8) for distribution to the subscribers, who are notified by executing Sub_C's callback function (9).

Publisher Contracts for the Same Event. Allowing more than one Pub_C for the same event type is dangerous since the new Publisher can accept data from a DO_O that the original Publisher would reject. To add an additional Pub_C safely, the initial Pub_C mediates access to its EM_C by other Pub_C. This approach provides a mechanism for expanding DO_O authentication mechanisms or policies and prevents the Pub_C from anticipating such mechanisms at deployment. For example, Figure 7.4 illustrates Publisher B's contract being added to Publisher A's EM_C. At deployment (1), instead of calling PS_C to create a new EM_C, Publisher B calls into Publisher A to request to be added to its EM_C (2). If the insurance company has authorized Publisher B (3), Publisher A adds the new publisher (4) and returns the address of its EM_C for Publisher B to use (5). Subsequent updates from Publisher B's DO_O (6) are sent to A's event manager (7), and distributed to the same list of subscribers (8).

Incentivizing Publications. In Decentagram, Sub_C rely on the EM_C to notify them of events by invoking their callback function, and the EM_C in turn relies on Pub_C to notify it when these

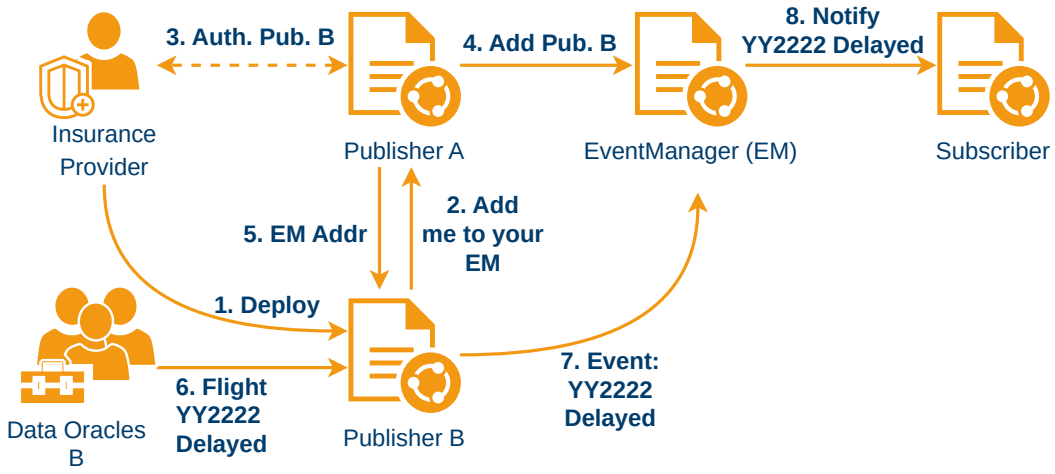


Figure 7.4: Registering a new Publisher to another Publisher's EventManager.

events happen. This series of cross-contract calls is initiated by a transaction from a DO_O, which invokes the Pub_C. Because these cross-contract calls happen within the same transaction, the transaction sender must include the cost of executing the target functions in the Pub_C, EM_C, and each Sub_C registered with the EM_C.

To ensure fair delivery of notifications, the EM_C requires each Pub_C call to include sufficient gas to execute all registered Sub_C callbacks. Without this check, the Pub_C could intentionally under-fund the call causing the transaction to be reverted or only a prefix of Sub_C to be notified. In either case, the Sub_O would be able to observe the transaction included in the block,[2] potentially allowing them to react to the event without Sub_C being notified.

The DO_O initiating the Publisher call is compensated for the cost of executing the transaction and successfully notifying the subscribing contracts by transferring the funds deposited by Sub_C at registration to the DO_O. The fee amount is specified by Pub_C during reg-

[2] A reverted transaction does not emit events, but the transaction (including the event data) would still be present in the committed block.

istration. If a Subscriber's deposit is insufficient, the Subscriber is removed from the list of subscribers before the callback is executed. The oracle may also be rewarded for each valid event submitted to the Pub_C, as specified by the Pub_C at deployment. For example, an application developer could offer a bug bounty program via Pub_C, where the first oracle to report evidence of a compromised component is rewarded. Our incentivization mechanism does not guarantee a particular fee schedule results in an incentive-compatible system since externalities may impact whether triggering an event is in the best interest of a DO_O. It does however provide system designers with a tool to compensate for such considerations.

Supporting Off-chain Subscribers The lower right portion of Figure 7.2 shows the overview of the off-chain portion of Decentagram. All nodes in a blockchain network have access to the blockchain data and the same view of confirmed blocks. The off-chain portion of Decentagram builds on this decentralized design by replicating the off-chain broker: each replica has the same copy of confirmed blockchain data.

Any application can subscribe to the events emitted by a Pub_C by authenticating the closest off-chain broker and subscribing to the events, specifying the address of the Publisher contract. The corresponding off-chain broker will be responsible for notifying the application when the relevant Pub_C emits an event.

Dealing with Chain Reorganizations A chain reorganization happens when two (or more) groups of miners in a blockchain network disagree on the sequence of blocks that form the canonical chain. Eventually, one sequence is extended enough to be accepted by the entire network, but the nodes that initially selected the abandoned chain must rollback the effects of

the abandoned blocks. Recent blockchain protocol designs such as proof-of-stake (PoS) are significantly less susceptible to chain reorganizations [87] than proof-of-work (PoW) protocols, but they are nevertheless possible under the right adversarial and network conditions [166]. Chain reorganizations could cause Sub_O in a pub/sub system to see and react to events in a sequence of blocks, only for those blocks to be rolled back and their transactions committed in a different order on the reorganized chain. Sub_C are unaffected by reorganizations since, like all contracts, they are reverted and re-executed on the new blocks.

An application may be indifferent to the order of events as long as they are eventually delivered (and not duplicated), but some may require a stricter ordering. While individual transactions cannot be replayed due to Ethereum's built-in per-address nonce for replay protection, enforcing a linear ordering of events from multiple oracles requires a nonce per event type, maintained by the contract and incremented for each event. The Publisher contract can require data oracles to include the current nonce. Since the publisher only accepts new events from an authenticated replica, malicious hosts cannot replay previous events by changing the nonce.

In the event of a reorganization, transactions from different oracles may be reordered in a way that causes a nonce to be invalid and dropped by the Publisher contract. In this case, the oracle should revert its own local state and retry the transaction when the nonce is once again valid. A consequence of the initial transaction being rejected is that a competing oracle may successfully publish an event with that nonce before the original oracle's second transaction succeeds. Once a block is finalized (currently after at most 64 blocks), it can no longer be reorganized, and is considered safe after at most 32 blocks. Luckily, block reorganizations are infrequent (a couple dozen out of over 7000 blocks per day), and almost always have a depth of

only one block [87].

7.1.3 Decentagram: Implementation

We implemented the Decentagram smart contracts described in Section 7.1.3 in Solidity, a language for the Ethereum blockchain. Solidity was chosen because of its maturity and rich set of features. Specifically, we used the setting of gas limits on cross-contract calls, function access controls, access to the transaction cost and value, and digital signature verifications in our on-chain Pub/sub service implementation. Off-chain, we relied on the logging mechanism provided by smart contract event emissions together with bloom filters, transaction receipts, and merkle trees supported by an Ethereum client to filter blocks for events to notify subscribers. All implementations for on-chain and off-chain components are available on GitHub at `https://github.com/lsd-ucsc/Decentagram`.

Decentagram Smart Contracts Figure 7.5 shows our implementation of the PubSubService and EventManager contracts. The core on-chain components of Decentagram are PS_C and EM_C, which create new event streams for Pub_C and register new Sub_C. An event registration happens when a publisher contract wants to create a new event stream to which it can publish, and it is done by calling the *register* (line 2) function of the PubSubService contract. If the publisher is unregistered, a new EventManager contract instance is spawned, with the publisher contract address being assigned as the owner of this new instance. By assigning ownership, we prevent malicious publishers from making unwanted publications to this EventManager. Once done, a *Registration* event is emitted signaling that a new event stream has been established and the

```
1       Contract PubSubService
2     on register() by sender
3       require sender is not registered
4       eventManager := EventManager(sender)
5       emMap[sender] := eventManager
6       emit Register(sender, eventManager)
7       return eventManager
8
9     on subscribe(publisher) by sender with value
10      require publisher is registered
11      eventManager := emMap[publisher]
12      eventManager.subscribe(sender, value)
13      return eventManager
14
15      Contract EventManager
16    on subscribe(subscriber, value)
17      require subscriber is not subscribed
18      require value >= τ
19      add subscriber to subList
20      subMap[subscriber] := value
21
22    on publish(data) by publisher from sender
23      require publisher in pubMap
24      gasNeeded = ϕ * len(subList)
25      require gasleft() >= gasNeeded
26      require not reentrantLock
27      reentrantLock = true
28      reimburse := 0
29      gasPrice := tx.gasPrice
30      foreach sub in subList
31        gasCanPay := sub.balance / gasPrice
32        gasCanPay := max(gasCanPay, ϕ)
33        startGas := gasleft()
34        try:
35          sub.onPublish{gas:gasCanPay}(data)
36        endGas := gasleft()
37        reimburse += (startGas - endGas) * α
38      emit Publish(data)
39      reentrantLock := false
40      send reimburse * gasPrice to sender
41
42    on addPublisher(newPublisher) by publisher
43      require publisher is owner
44      add newPublisher to pubMap
```

Figure 7.5: Pseudocode of the PubSubService and EventManager Contracts

address of the EventManager is returned to the publisher. Emitting a registration event allows the off-chain broker to learn of new event streams, including the publishers and EventManagers for these streams. The usage of this event will be explained in Section 7.1.3.

A Subscriber that wants to subscribe to a publisher's event stream calls the PubSub-

Service's *subscribe* function with the publisher's address as the argument. The PubSubService first checks that the publisher is registered, then it retrieves the EventManager associated with that publisher and calls the EventManager's *subscribe* function, passing in the amount of Ether that the subscriber included with the transaction (line 12). Before adding the subscriber, the EventManager checks that the subscriber is not already subscribed to the event stream and that the value passed in meets the minimum deposit requirement (τ). The EventManager adds the subscriber and the corresponding balance before returning back to the PubSubService, which will return the address of the EventManager to the subscriber. The subscriber can then use this address to authenticate the caller of their callback function, and filter out callbacks from sources other than the EventManager it subscribed to.

On the publisher side, an event publication is originated from a transaction that invokes one of functions in publisher smart contract, which will possibly trigger a state update. When the conditions to publish the events are met (e.g., some state has been updated), the function in the Publisher contract can call the EventManager's *publish* function, passing in the data relating to the event (line 22). Notice that there is only a single parameter of type *bytes* to represent all the data pertaining to the event. This not only simplifies the implementation of the EventManager, but also provides flexibility in how publishers and subscribers can deal with events for different types. For each type of event, a common interface can be defined by the application to allow publishers to encode event data and subscribers to decode the data.

Before notifying the subscribers, the publish function first checks that the caller of the function is a registered publisher contract (line 23) and that there is a sufficient amount of gas left in the transaction to notify all the subscribers in the list (line 25). This value is

calculated by multiplying the gas limit the contract set for each subscriber's callback function (ϕ) by the number of subscribers. A lock is also used to prevent reentrancy attacks (line 27), which has been shown to be a critical security risk in smart contracts [59]. If the requirements are met, then the function continues and iterates over the list of subscribers to notify each subscriber of the new event. We calculate the amount of gas taken by the callback function by wrapping the call between two calls to the built-in `gasleft` function provided by Solidity, and use this amount to add to the reimbursement for the transaction sender (lines 37). The calculated amount is multiplied by the transaction's gas price (α) to make sure that the reimbursement is accurate. After all of the subscribers have been notified, the reentrancy lock is released, an event is emitted, and the reimbursement amount is transferred to the transaction sender from the EventManager contract's balance.

In the previous section, we discussed how an EventManager can support multiple publishers if the owner of the EventManager allows it. If a publisher, A, supports this feature, it will expose a function that another publisher, B, can call to request to be added. Once approved, the function will make the call to the EventManager's *addPublisher* function (line 42). The EventManager will first check that publisher A is the owner, and then add publisher B, after which it will accept publications from B.

Securing against Problematic Subscribers. Subscribing contracts are untrusted, so we must isolate the execution of on-chain subscriber callbacks to prevent a buggy or malicious callback from causing an entire event notification to fail. All callback invocations are wrapped in *try-catch* blocks (exceptions are ignored), and the gas usage of the call is calculated after the call

returns or an exception is caught.

To prevent gas exhaustion attacks [18], the EM_C limits gas usage by callback functions. If gas usage exceeds the remaining gas in the Sub_C's balance or a predefined gas limit set by the Publisher contract, the function is interrupted, and the EM_C continues notifying other Sub_C.

On-chain Subscribers and the Gas Limit. Blocks in Ethereum have a size limit that is defined by the maximum amount of computation required to execute the transactions contained in the block. Currently, the gas limit is 30 million [218]; a single transaction using as much as 30 million gas would take up the entire block. Our EM_C contract limits the amount of gas used by callback functions, but the overall transaction gas limit constrains the number of on-chain subscribers a specific EM_C can have.

The gas limit for callback functions is set by the (initial) Pub_C when a new EM_C is created. For reference, the cost to make cross-contract calls (such as to subscriber callback functions) is around 3347 gas, the cost to set a persistent (stored in the contract) boolean flag is 3050 gas, and the cost to add an element to a hash map is 22,200 gas. Assuming Pub_C consumes 2 million gas (the worst case when authenticating a new data source), this leaves 28 million gas remaining as an upper bound for executing subscriber callbacks. Balancing the tradeoff between the number of Sub_C and the callback gas limit is left to Pub_C. Our default gas limit per subscriber is set to 200,000, which allows for about 140 Sub_C.

One way of surpassing the maximum number of Sub_C would be to split subscriber notifications over multiple transactions. The publisher could register and deploy an additional

EM_C for each new Sub_C, and notify each EM_C in separate transactions, each with a separate gas limit. The drawback of this approach is that some Sub_C will be notified later than others, and it may be necessary to create additional incentives to ensure the transaction sender notifies all Sub_C (e.g., requiring an up-front deposit that covers the total notification cost).

Only Sub_C actions triggered by a Pub_C transaction through a callback function are subject to these gas limits. For M&R style workflows, a user can use Sub_O to monitor the on-chain events. When a new event is received, the Sub_O relays that event to the desired smart contract in a subsequent transaction. Even more effective is a hybrid approach where a minimal Sub_C performs critical updates to the Sub_C's state, such as revoking access or setting flags. Following this, the Sub_O can send a transaction to complete any remaining tasks related to the event. For example, invoking a callback that just sets a boolean flag costs about 6500 gas. An EM_C with a subscriber gas limit of 6500 can support up to around 4300 Sub_C.

Sending Events to Off-Chain Subscribers Figure 7.6 shows an overview of the off-chain broker's workflow. The Geth Client [81] communicates with other Ethereum nodes to maintain the blockchain data locally. The off-chain broker constantly polls the Geth client for new block headers and checks for new events emitted by the addresses of EM_C contracts it is monitoring. When a new event is received, the off-chain broker consults its list of Sub_O for that event type and sends each Sub_O the new event.

The blockchain data is usually generated and stored by the Ethereum nodes in the network. Different nodes in the network communicate to synchronize their view of the chain. Instead of implementing these logics in the off-chain broker, we directly retrieve the blockchain

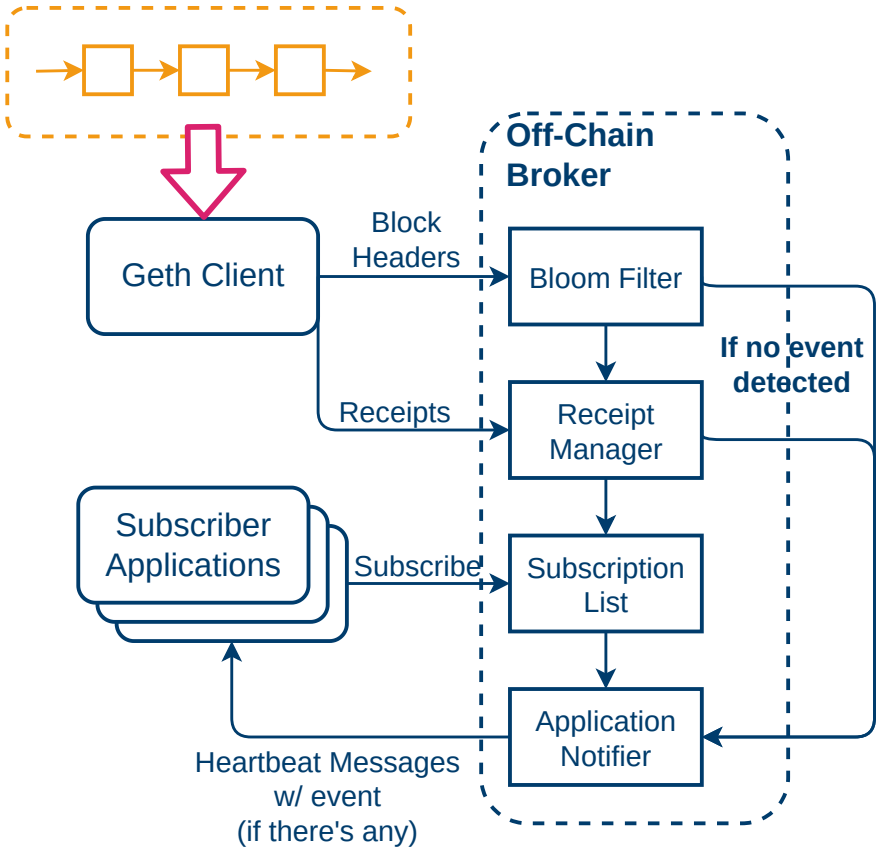


Figure 7.6: Overview of the off-chain broker Implementation

data from the Geth client using RPC calls via TCP connections. Therefore, the Geth client is responsible for maintaining the blockchain data and synchronizing with other nodes in the network. Alternatively, developers may choose to use other Ethereum clients that follow the standard Ethereum protocol and provide compatible RPC interfaces.

On-Chain Events. After receiving blockchain data, the off-chain broker checks for monitored contract events. Ethereum contracts may emit named tuples called (helpfully) *events*, which are included in the transaction receipt. Decentagram uses these events to communicate with off-chain brokers.

The constructor of PS_C emits a `Deploy` event marking the initialization of the on-chain broker. The off-chain broker then begins to monitor for `Register` events. Each time a publisher registers, a `Register` event announces the address of its corresponding EM_C. With this information, the off-chain broker can maintain a mapping of publisher addresses to event manager addresses, and start to monitor for `Publish` events, in order to know the occurrence of events emitted by that event manager. By knowing the address of PS_C, the off-chain broker can determine the event manager addresses it wants to monitor for `Publish` events.

These events emitted by the on-chain contract will then be visible in the log data of transaction receipts. A naive approach to monitor these events will be retrieving all the receipts of every block and checking their logs to see if any of them contain the events of interest. However, parsing and reading all the receipts incurs a significant amount of overhead, especially when validating the receipts root hash is necessary. We discuss more about this overhead in our evaluation of the off-chain broker in Section 7.1.5. Instead, we can look at the bloom filter contained in the block header to probabilistically determine if a block contains events we are interested in.

In Ethereum [214], each block header contains a bloom filter that is constructed using the address of contracts that have emitted event(s), the event's signature, and indexed event's arguments. To efficiently filter events, the Broker only has to check the header bloom against the contract addresses and the event signatures. If the filter signals a positive result, the rlp-encoded receipts will be fetched and processed. False positives can occur, but the false positive rate on Ethereum's main chain is promisingly low (about 0.5% [185]).

Subscriber Services. The off-chain broker could start at any point of time. Some instances may be started before PS_C is deployed, while other instances may be started after deployment and some events have already been published by EM_C. Similarly, the Sub_O may start subscribing to the Pub_C before or after the Pub_C has published any events. For instance, a Sub_O wants to subscribe to a Pub_C that emits events when a new item is added to a revocation list. If the Sub_O starts subscribing after Pub_C has already published some events, the Sub_O will also want to know what items are already in the revocation list. Because of this, the off-chain broker must maintain a record of all the events that have been published since the deployment of PS_C.

As described above, the off-chain broker knows the beginning of the on-chain Pub/sub services and all the addresses of the existing event manager contracts. During the boot phase of the off-chain broker, it will retrieve history blocks from the Geth client, and record all the past events. So, in addition to notifying the Sub_O of the new events, the off-chain broker will also log these events in storage. At runtime, the off-chain broker accepts event requests from Sub_O, which specify the address of Pub_C at the beginning of a TCP connection. The Broker replies with all the past events and corresponding block numbers from that Pub_C. The Sub_O receives new events over the TCP connection as they occur.

To notify the Sub_O of the new events, the off-chain broker can simply send the event data via the opened TCP connection. However, this approach is not secure enough in case the network connection between the Broker and the Sub_O is untrusted. For instance, an adversary may suppress the TCP connection between them to pretend that the publisher has not published any new events. A subscriber that is listening to the flight delayed events from the off-chain broker may be misled to believe that the publisher has not published any delayed flight since

the subscription.

To address this issue, the Broker will send heartbeat messages through the opened TCP connection periodically. Each heartbeat message contains the latest block number of the blockchain. In case a new event is published, the event data will be sent along with the heartbeat message. If the Sub_O does not receive any heartbeat message for a certain period of time, it may assume that the connection is broken, and take appropriate actions to prevent possible losses.

7.1.4 Detecting Faults On-Chain

We now present three types of Publisher contracts that are able to validate digital signatures of off-chain components and detect faults: the *Voting Publisher*, the *conflicting message Publisher*, and the *compromised key Publisher*. It should be noted that fault detection is not limited to just these three types, and applications can define their own fault detection mechanisms based on the data that can be verified. For these fault detection contracts, we consider a replicated system consisting of a set of replicas, each of which can be uniquely identified by a public key, with the corresponding private key used to sign messages that they send. A fault detection contract maintains a list of public keys, one for each replica, as well as a list of replicas that have been detected as faulty. The list of faulty replicas initially starts out as empty and is updated as replicas are added to the list based on the contract's fault detection mechanism, which we explain below.

Voting Publisher. The *Voting Publisher* is similar to the propose-and-vote scheme used in CCF [184], but here, we use it to vote out replicas instead of stakeholders. During runtime, any

of the replicas that have experienced subjectively faulty behavior locally can broadcast their suspicion to the other replicas and send a transaction to the *Voting Publisher* contract. As we explained in Section 3.3.1, subjective faults cannot be proven to a third party, so a single vote alone is not enough to convince a third party of the malicious behavior; hence,the most a replica can do is to relay its suspicion. After incrementing the vote count, the contract will check if the number of votes has reached a threshold (e.g., $\lfloor \frac{n-1}{3} \rfloor$ for BFT systems); if so, the contract will add the replica to the detected list and notify the event manager to publish the corresponding detection event.

Conflicting Message Publisher. For situations where replica equivocations are signed, the *Voting Publisher* is inefficient since a set of these conflicting signed messages is enough to prove malicious behavior. In these situations, the *Conflicting Message Publisher* can be used to detect replicas and quickly add them to the detected list without waiting for votes from other replicas. A replica i that has received conflicting messages from another replica j can send the hash and signature for the two messages to the contract, which will then check that the hashes and signatures are valid before adding j to the detected list.

Compromised Key Publisher. Another way that a replica can be deemed faulty is if the private key associated with that replica has been compromised. In Section 3.6, we mentioned that some BFT protocols deal with accumulated faults by means of proactive recovery, where replicas are rebooted and their keys are refreshed. Rekeying is a way to relinquish control of a replica from an adversary that has compromised its key and is using it to communicate with other replicas. The *Compromised Key Publisher* provides a way to detect these replicas whose

keys have been compromised. Any replica (or client) that has detected that a replica's key has been compromised can use this key to sign a predefined message (e.g., "REVOKE THIS KEY") and send it to the contract. By verifying the signature of this message, the contract can detect that the replica is faulty not only when its private key is completely exposed but also when the private key is partially exposed, such that it is sufficient for any party to forge a signature.

7.1.5 Evaluation

Publisher, Subscriber, and On-Chain Broker Contracts We evaluate the gas cost of our implementation of the on-chain broker with minimal publisher and subscriber contracts. Pub_C registers with PS_C and exposes a function to publish a fixed-payload event. Sub_C subscribes to Pub_C events and verifies it receives the expected payload for each event. To evaluate gas cost, we deployed our contracts on Ganache, a local Ethereum blockchain testing environment [203].

Based on the transaction receipt, the gas cost of deploying PS_C is 623,330. In addition, we evaluated the gas costs of *using* the on-chain broker, including three major operations - registering, subscribing, and publishing. In this experiment, we measured the cost of each operation, and repeated the process with the number of publishers and subscribers increasing from 1 to 20. For event registration, we calculated the average gas used per publisher. As the number of on-chain publishers increases, the average gas used per publisher stays relatively constant, at around 570 thousand gas, with negligible fluctuations (less than 50 gas). Ethereum's mapping data structure allows data to be fetched and stored with constant gas cost, so the gas cost of managing the address of Pub_C and their EM_C does not increase with the map size.

A similar process is used to evaluate the gas cost of subscribing. We deployed the

Sub_C, with each of them subscribing to a different Pub_C, and recorded the gas used. As the number of Sub_C increases, the average gas used per subscriber also stays constant, at around 160 thousand gas, with negligible fluctuations (less than 10 gas). Like the `register` operation, the `subscribe` operation uses the mapping data structure to look up the address of the event manager, which results in a constant gas cost.

Next, we evaluated the cost of publishing by deploying multiple Sub_C that subscribe to a single Pub_C. With one subscriber, the publishing cost is 134,768 gas. As the number of Sub_C increases, publishing cost increases linearly, with the marginal cost around 76,000 gas for each additional subscriber. That is because the `publish` operation iterates through the list of Sub_C to invoke their callback functions. Today, the cost per gas in the Ethereum Mainnet is around 0.001 cents. Thus, the base cost to publish a message is \$1.35 increasing by \$0.76 for each Sub_C. Our contracts are also deployable on EVM-compatible chains. We have deployed and tested Decentagram contracts on Avalanche as well as "Layer 2" (L2) chains Polygon, BNB, and Optimism. L2 chains execute contracts and transactions on a local chain and commit checkpoints to the main Ethereum chain. For L2 chains, off-chain components in Figure 7.2 would monitor and interact directly with the L2 chain.

Deploying Decentagram to such a chain greatly reduces the cost to publish messages while still benefiting from the security of the Ethereum mainnet. For example, the cost per gas in Polygon network is $4 \cdot 10^{-6}$ cents, and $6 \cdot 10^{-5}$ cents in BNB network. At these prices, the (base, per-subscriber) cost is (\$0.0053, \$0.0030) and (\$0.086, \$0.049), respectively. Note that subscriber fees are expected to reimburse publishing costs, and these prices represent the "break-even" cost for publishers. Higher transaction fees require higher subscriber fees to in-

Operations	Digital Signatures (secp256k1)
Fault Detection by Voting	
Deploy	852,279
Vote (average)	81,267
Detecting Conflicting Messages	
Deploy	814,601
Report	70,047
Detecting Compromised Keys	
Deploy	762,302
Report	66,015

Table 7.2: Gas Costs for fault detection Contracts

centivize publisher participation, so reducing these fees enables a wider range of Decentagram applications.

Fault Detection Contracts Next, we evaluate the gas costs for the fault detection contracts; the results are shown in Table 7.2. As we can see, the amount of gas to deploy all three types of contracts are similar, since most of the cost comes from subscribing to the PubSubService and retrieving a new instance of an EventManager contract through which it can publish. Likewise, the difference in cost to send a transaction to report a faulty replica differs only slightly between the three types of contracts. Note that the cost to verify a *vote* is higher than the cost to detect conflicting messages. This is because in the *Voting Publisher*, additional bookkeeping is needed to check and record each replica's vote as well as tallying the total number of votes.

Off-chain Block Processing In the off-chain evaluation, we focus on block processing efficiency, so that the Sub_O can be notified of the events in these blocks in a timely manner. Among those steps described in Section 7.1.2, we have identified the major overhead in block process-

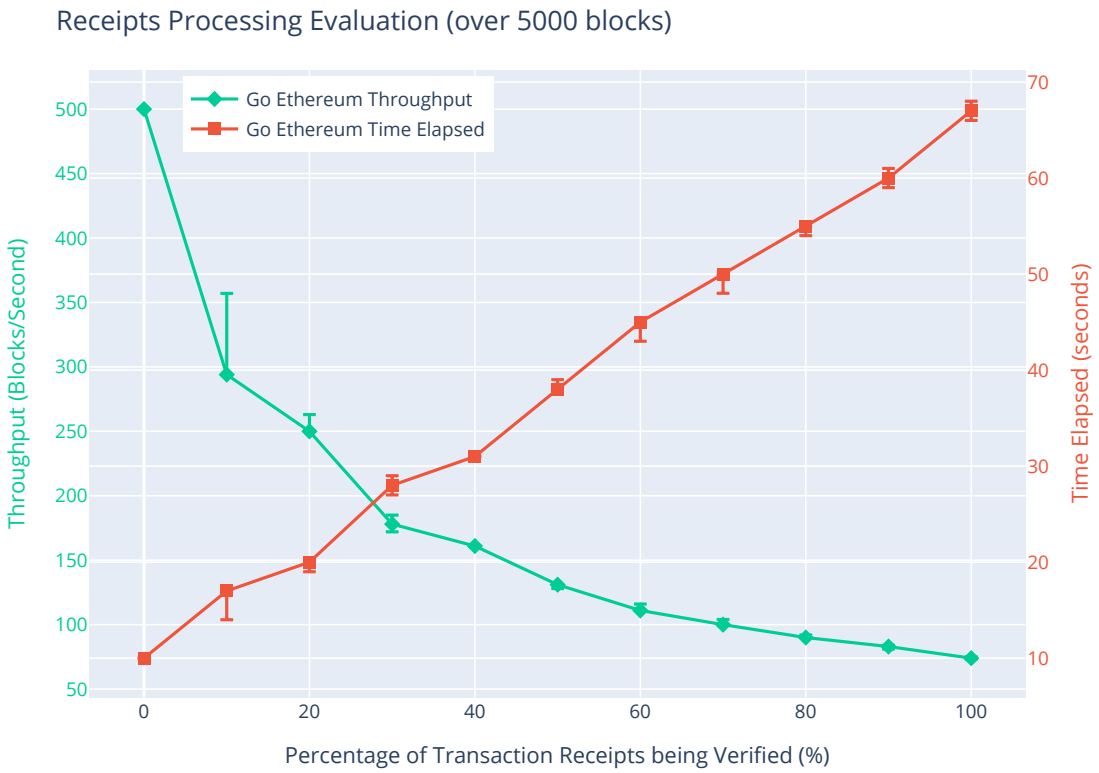


Figure 7.7: Off-chain Receipts Processing Evaluation. The test was conducted three times, and the error bars represent the maximum and minimum values, however, most of the differences are too small to be visible in the figure.

ing as being the parsing and validating of transaction receipts. The experiment is conducted on a PC running the Ubuntu operating system, equipped with an i3-7100T CPU, and 32G of RAM. The results are shown in Figure 7.7.

The off-chain broker uses the bloom filter in the block header to skip processing of blocks that return a negative result. In our experiment, we simulate positive bloom filter results for 0% to 100% of 5000 blocks in 10% increments. These blocks were selected from the range 8,875,000 to 8,880,000, in the Ethereum Goerli testnet. We can see that the time taken by both implementations increases linearly as the possibility increases. Even when receipts from every

	$\mathbf{DO}_O \rightarrow \mathbf{Pub}_C$	$\mathbf{Pub}_C \rightarrow \mathbf{Sub}_C$	$\mathbf{DO}_O \rightarrow \mathbf{Sub}_C$
Decentagram	12 s (Max:35, Min:5)	0 s (Max:0, Min:0)	12 s (Max:35, Min:5)
M&R	12 s (Max:46, Min:10)	12.5 s (Max:35, Min:9)	25 s (Max:58,Min:22)

Table 7.3: Median End-to-End Latency Comparison.

block have to be parsed and verified, the throughput of Geth is 74.63 blocks/second, which is around 896 times faster than the block arrival rate. Hence, regardless of the off-chain broker version the Sub_O uses, there should not be any backlog of new blocks, and Sub_O will be notified of new events in a timely manner.

End-to-End Latency Comparison Decentagram only requires that a candidate event be signed by an authenticated component, avoiding the need for an off-chain consensus round prior to publication as in Chainlink and Chios. Therefore, we evaluate the latency of Decentagram by measuring the time from when events are published to when they are delivered. The M&R approach serves as a good baseline for comparison since it also does not require consensus prior to publication. To compare the efficiency of Decentagram with the traditional M&R approach, we conducted an experiment to measure the end-to-end latency of notification delivery. In both test cases, a DO_O publishes new data to a Pub_C, and a Sub_C waits to be notified of the new data. Additionally, a Sub_O monitors new blocks to be notified of the new data as well as the confirmation from the Sub_C showing that it has processed the data. The Ethereum Goerli testnet, which has a block arrival rate of 12 seconds/block, was used in this experiment; and all the test cases were repeated 20 times, with the median, minimum, and maximum values reported in Table 7.3. The first column shows the time elapsed from DO_O publishes data until Sub_O receives it via the

event emitted by Pub_C. The result between the two approaches are almost the same, since both of them require the DO_O to make a transaction to Pub_C, and then Sub_O will be notified when they receive the event. The second column shows the time required for the Sub_O to receive the subsequent confirmation from Sub_C after it has been notified of the event from Pub_C.

As expected, the difference in latency between the two approaches is significant. With Decentagram, the Pub_C was able to notify the Sub_C via cross-contract call within the same transaction. Thus, when Sub_O received the event, it also received the confirmation that Sub_C has finished processing the data. While in the M&R approach, the Sub_O has to react to the event by making another transaction to Sub_C, which results in a longer latency. The third column shows the total time elapsed from event publication to the confirmation from Sub_C. Decentagram is able to complete the entire process using only one transaction, reducing the latency by half.

During the experiment, we encountered network fluctuations, with some time slots being skipped causing the new block to arrive later than expected, and some blocks being empty causing our transactions to be delayed until the next block. The time it takes for data to propagate from DO_O to Sub_C took even longer when these two situations occurred simultaneously. Such situations are common on the testnet but rare on the mainnet. However, the mainnet is also more congested, which would also cause similar effects. In both cases, Decentagram is less affected by these fluctuations since it only requires one transaction to notify subscriber contracts compared to two transactions required by the M&R approach.

Application Domain Topics in Decentagram are identified by the Pub_C address, so the number of channels is not limited in any practical way (there are 2^{160} distinct contract addresses).

The number of Sub_C addresses per channel is capped by the block gas limit (see Section 7.1.3). In our default configuration, each channel can support up to 140 on-chain subscribers. Note this limit only applies to *on-chain* subscribers—there are no limitations on the number of *off-chain* subscribers. In fact, since the off-chain brokers are decentralized, the number of off-chain subscribers scales indefinitely.

Message delivery latency relies on the block arrival rate of the underlying blockchain network. The 12 seconds/block rate is the result of our use of the Ethereum blockchain. EVM-compatible L2 networks such as Polygon or BNB have faster block rates (2 or 3 seconds, respectively). Taking these aspects into account, Decentagram is most suitable for applications that have event-generation rates on the order of seconds, need to scale to massive numbers of off-chain subscribers, with on-chain subscribers reasonably distributed over many topics.

7.1.6 Decentagram: Application to Phoenix

Earlier in the chapter, we discussed how trust in the CM, as well as the lack of specification for its liveness property, are issues that can prevent Phoenix from being implemented in permissioned blockchain systems, even though it is already used in traditional replicated systems. In this section, we will show how we can apply the concepts behind Decentagram to implement the CM as a smart contract that will not only address these issues but also provide the CM with a richer set of fault detection mechanisms. Naturally, by having the CM as a smart contract, it inherits the safety and liveness properties of the underlying blockchain system. Specifically, it will have a stronger liveness property than it would have if it were to be replicated over a set of replicas using a BFT protocol since that would only be able to tolerate

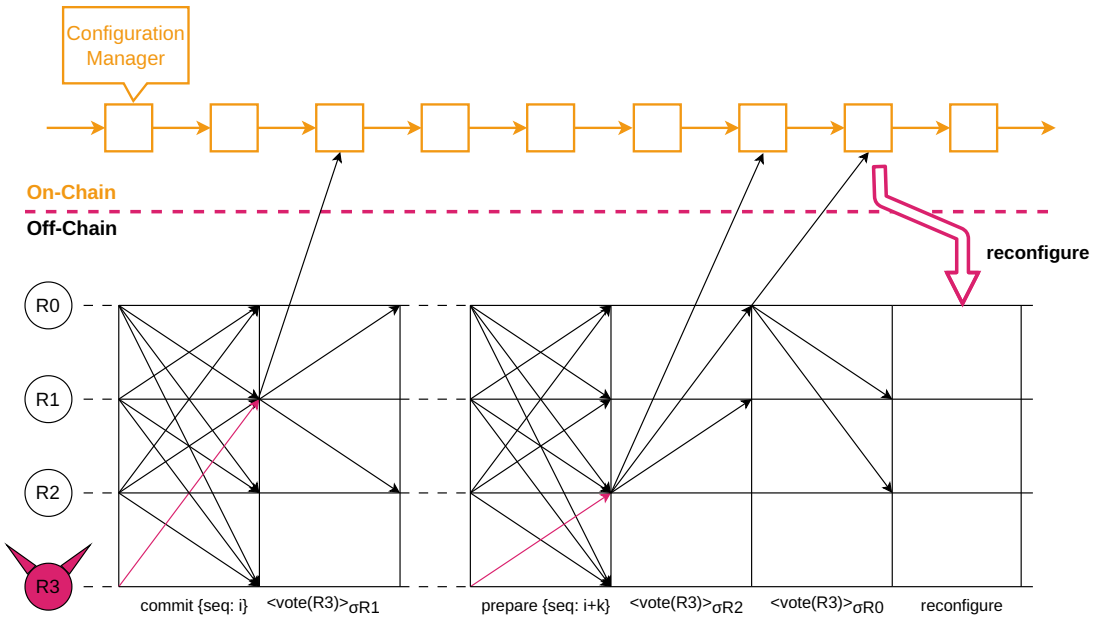


Figure 7.8: Phoenix with the CM as a Voting Publisher in a permissioned blockchain system.

up to less than a third of the replicas being faulty, whereas the smart contract can tolerate up to half of the miners being faulty (in a PoW system).

First, let us consider the case of having the CM as a Publisher for a system that is using Phoenix as described in Section 3. For simplicity, we can disregard the PubSubService, EventManager, as well as the Subscriber contracts, and have the CM directly emit *reconfiguration* events when enough votes have been collected. On the off-chain side, replicas will act as both a publisher that will send (publish) their votes to the CM as well as the subscriber that monitors the blockchain for reconfiguration events. Figure 7.8 shows a modified example from Section 3.3.1 in which replicas will send their votes to the CM to trigger a reconfiguration and replace a Byzantine replica. One difference between the CM and the *Voting Publisher* described in Section 7.1.4 is that in addition to keeping track of a list of replicas in the current active replica set,

it has another list for backup replicas that will be used as replacement. Once again, we have the Byzantine replica $R3$ exhibiting faulty behavior towards correct replicas $R1$ and $R2$ over time, causing these replicas to send their votes against $R3$ to the CM (in transactions that make it to separate blocks). The CM simply verifies these digitally-signed *vote* messages before adding them to a tally against $R3$. After replica $R0$ sends its own vote after receiving *vote* messages from $R1$ and $R2$, the CM concludes the voting round, then constructs a new configuration to replace $R3$ with a replica from its backup list, and emits a *reconfiguration* event. When $R0$, the leader, notices the reconfiguration event in a block, it will include the new configuration within this event in a *propose* message to be sent to the other replicas.

7.1.7 Decentagram: Application to Payment Channels

```
1   Contract PaymentChannel
2     on constructor(bals, deps, withdrawwait)
3       [...]
4       lock := false
5       malicious := ∅
6       eventManager := PubSubService.register()
7
8     on update(bals, round) by sender
9       [...]
10
11    on withdraw(bals) by sender
12      [...]
13
14    on notify(data) by eventManager
15      address := decode(data)
16      if address in balances
17        lock := true
18        malicious := address
19
20    on withdrawExit(bals, round, deposit) by sender
21      if (lock and malicious != sender) or
22          closingparty == ∅
23            [...]
```

Figure 7.9: A payment channel contract as an on-chain subscriber.

Now, let us revisit the payment channel construction from chapter 6, and explore how

to make it even more robust with Decentagram. As we mentioned, the ability of Decentagram to instantly notify on-chain subscribers of events can be used to detect a malicious party whose address has been blacklisted and allow a payment channel contract to lock itself to prevent the malicious party from making the first move to close the channel. The revised payment channel contract is shown in Figure 7.9, which contains three minor changes. First, in the constructor, we declare a lock as well as a variable to assign the address of the malicious party. The constructor also makes a call to the *PubSubService* contract to register itself to be notified of events. Second, we define a new function *notify* that is a callback function that can only be invoked by the *EventManager* to notify the payment channel of new events. The *notify* function takes only a single parameter because, as mentioned in Section 7.1.3, the data format varies between event types; in this case, the data between the Publisher and Subscriber is in the form of an address. Thus, this data parameter is decoded into an address that is then checked by the payment channel to determine if it is a party in the channel. If so, the contract is locked, and the address is assigned to the malicious party variable. Finally, we add a condition to the *withdrawExit* function to prevent the malicious party from closing the channel if the lock is set and it is the sender. With these changes, a malicious party that has been blacklisted will not be able to make the first move in closing the channel but will still allow both parties to close if they are honest.

7.1.8 Summary

In this chapter, we presented the Decentagram framework that allows for instant dissemination of events on-chain and timely event notification off-chain using the pub/sub mes-

saging model. Our construction of the on-chain Publisher, Subscriber, and Broker contracts is resilient to Byzantine failures and provides incentives for event publications. We showed that the ability to verify digital signatures on-chain provided by Decentagram can be used in various ways to facilitate on-chain fault detection and that our work in Phoenix, described in Chapter 3, can leverage this mechanism to not only detect Byzantine replicas but encode its configuration manager as a smart contract to provide trustless reconfiguration. Then, we presented a revision of our robust payment channel construction, described in Chapter 6, that made use of Decentagram's instant on-chain notification using cross-contract calls to enable a locking mechanism that prevents a party from closing the payment channel using an old state if it has been publicly exposed to be malicious. We evaluated Decentagram by measuring the gas cost to deploy the on-chain contracts and execute their functions. Additionally, we evaluated the receipts processing throughput of the off-chain broker, showing that it is able to process new blocks for events at a rate much faster than new blocks can be mined, so clients are able to receive event notifications in a timely manner.

7.2 Smart Contract as a Heartbeat Failure Detector

In this section, we describe how we can enhance smart contracts to be used as a heartbeat failure detector, allowing unresponsive processes in a service to be detective and penalized. Heartbeat failure detectors [60] rely on the use of a timer variable (T), requiring that a process sends a heartbeat message within the time period specified by T before it is detected as faulty. Although this approach is effective in detecting crashes, it is generally ineffective in detecting Byzantine faults since the faulty process can respond normally to these heartbeat messages but deviate from the protocol in other ways, as discussed in Chapter 3. For blockchain applications in which untrusted (and possibly Byzantine) processes communicate with each other, encoding this type of failure detector at the processes themselves is unreliable, since a Byzantine process can choose which processes to respond to and which to ignore, and it is impossible to prove that the Byzantine process omitted a response. However, we can make heartbeat failure detectors in a blockchain application by encoding them in smart contracts, along with the on-chain part of the application.

The underlying consensus protocols in blockchains enforce a block arrival rate, and any client that monitors the blockchain can expect that the average time new blocks arrive will be within the time defined by the block arrival rate. This means that if we encode the heartbeat interval T in a smart contract in the number of blocks, we can compel a process to respond within T blocks in order for it not to be considered faulty. For instance, consider two processes p and q form a service agreement in a smart contract, with both having to place a security deposit in the contract as a form of compensation when either party fails to meet their end of the

agreement. If p, as the service provider, fails to respond to q's request, q can invoke a method in the contract to trigger T to start, and if p does not respond after T blocks, the contract will automatically penalize p by subtracting from its security deposit. It could be the case, however, that p intentionally does not respond within T blocks, but any rational p would not do this since it would result in monetary loss. With this in mind, we propose a blockchain-based FaaS platform that uses this approach to on-chain failure detection, leveraging the high availability guarantee of blockchains to enforce heartbeat responses from service providers and provide a better quality of service for clients.

7.2.1 Heartbeat Failure Detector Smart Contract Design

Figure 7.10 shows the design of a smart contract that encodes a heartbeat failure detector for unresponsive service providers. The contract encodes a service agreement between a client and a provider that specifies the amount that the client will pay the provider for the services that it provides. When the contract is deployed using the `constructor`, the balances and addresses of the provider and client are initialized, along with the security deposit from the provider and the fee that the client pays each time it invokes the service from the provider. The provider can claim the service fee by invoking the `claim` function, providing a proof that the service was provided. This proof can be a digitally signed message from the provider containing the signed request from the client and the response from the provider containing the results of that request. If the proof is valid, the fee will be subtracted from the client's balance and added to the provider's balance.

The service provider could become unresponsive due to it going offline or maliciously

```
Contract ServiceAgreement
  on constructor(bals, addrs, dep, serviceFee)
    timeout := Δ
    disputePenalty := ρ
    providerBal := bals[0]
    providerAddr := addrs[0]
    clientBal := bals[1]
    clientAddr := addrs[1]
    securityDeposit := dep

  on claim(serviceProof) by provider
    if isValidProof(serviceProof)
      if isSufficientFunds(clientBal)
        providerBal += serviceFee
        clientBal -= serviceFee

  on deposit() by client
    clientBal += msg.value

  on serviceDispute(requestId) by client
    if isValidRequest(requestId) and !dispute.resolved
      dispute.id := requestId
      dispute.startBlock := block.number
      dispute.resolved := false
      emit ClientDispute(dispute.id)

  on disputeResponse(requestId, serviceProof) by provider
    if isValidRequest(requestId)
      if isValidProof(serviceProof)
        currentBlock := block.number
        if currentBlock - dispute.startBlock <= timeout
          dispute.resolved := true
          emit DisputeResolved(dispute.id, SUCCESS)

  on unresolvedDispute(requestId) by client
    if isValidRequest(requestId) and !dispute.resolved
      currentBlock := block.number
      if currentBlock - dispute.startBlock > timeout
        securityDeposit -= disputePenalty
        clientBal += disputePenalty
        dispute.resolved := true
        emit DisputeResolved(dispute.id, FAILURE)

  on closeContract() by sender
    require(msg.sender == providerAddr or msg.sender == clientAddr)
    send providerBal to providerAddr
    send clientBal to clientAddr
```

Figure 7.10: A Service agreement contract between client and provider with a dispute mechanism.

withholding responses from the client. In this case, the client can try to force a response from the provider for its request by invoking the `serviceDispute` function. If the request is valid, the dispute process starts, and the contract sets a timer for the provider to respond, starting at

the current block number. A `ClientDispute` event is emitted to let both the client and provider know of the dispute and give the provider a chance to respond to this dispute. To respond, the provider can invoke the `disputeResponse` function with the requestId and a proof that the request was successfully processed. A malicious will be inclined to respond to the dispute since an unresolved dispute will result in a penalty fee being subtracted from the provider's security deposit. An unresponsive provider that has crashed will also result in the client dispute being successful because it wouldn't be able to respond to the dispute within the timeout period, in which case the detection mechanism from the smart contract will be successful in detecting the crashed provider. When the provider fails to respond within the timeout period, the client can invoke the `unresolvedDispute` function to resolve the dispute and claim the penalty fee from the provider's security deposit. Finally, the client or provider can close the contract by invoking the `closeContract` function, which will send the remaining balances to their respective addresses.

7.2.2 Applications of On-chain Heartbeat Failure Detection

The service agreement contract described in the previous section can be used in any application that requires a service agreement between two parties to be encoded in a smart contract. One example of such an application is Function-as-a-Service (FaaS). FaaS, also known as serverless computing, is a cloud computing paradigm that has been gaining popularity, with many cloud providers offering FaaS services, such as AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. In this model, clients are only charged per invocation for the amount of resources used to execute their workloads, as opposed to the standard Infrastructure-

as-a-service (IaaS) model where clients are charged for the entire duration in which their resources are reserved, including when the resources are idle. Recent works have aimed to adopt the FaaS model in blockchain-based applications [17, 100, 99, 95] to utilize the resources of a decentralized network of nodes while also providing availability, integrity, and Byzantine fault tolerance. Using our service agreement contract with heartbeat failure detection, a blockchain-based FaaS platform can provide a more reliable service to clients by helping clients determine if unresponsive service providers are malicious or have actually crashed.

Chapter 8

Conclusion

Byzantine fault tolerant protocols have once again become a topic of great interest with the recent adoption into blockchain systems and the resulting widespread popularity and usage of blockchain-based by users all around the world. The existence of Byzantine faults and the damage that they can do if left unchecked means that it is critical for BFT systems to not only tolerate faults but also detect and remove them. This dissertation explores the ways in which fault detection is done in both permissioned BFT systems and permissionless BFT systems and investigates the possibility of enhancing existing fault detection techniques to make the underlying system more robust against failures. In this chapter, we summarize the contributions of this dissertation.

8.1 Fault detection in permissioned BFT systems.

Permissioned BFT systems make the fundamental tradeoff of prioritizing strong consistency over availability in the presence of network partitions. This means that undetected

faults in prolonged service deployment can accumulate and eventually surpass the predefined fault tolerance threshold of the system, allowing a Byzantine replica to disrupt the system without the need to be able to control other replicas or message delivery over the network, as described in literature. Fault detection in permissioned BFT systems is a well-studied problem that has resulted in a plethora of works that looked into ways replicas can misbehave and how to detect them. Ultimately, these fault detection mechanisms are not used to their full potential in that they do not provide a way to deal with faulty replicas even if they are detected. We presented the design and implementation of Phoenix, a reactive reconfiguration protocol that can leverage these fault detection mechanisms to drive reconfiguration, allowing the system to detect and remove faults as they occur. By replacing faulty replicas with new ones, we essentially reset the number of accumulated faults, making the deployment of the system more realistic since it would be able to tolerate an unbounded number of faults in a prolonged deployment, so long as the number of simultaneous faults abide by the fault threshold. Prior BFT protocols that include a reconfiguration mechanism make an unrealistic assumption that the configuration manager, the entity that initiates reconfiguration, knows which replicas in the system are faulty. Using Phoenix, the CM does not have to make such an assumption and can instead rely on localized fault detection information from the replicas to make better reconfiguration choices. Our evaluation of Phoenix shows that the reconfiguration protocol introduces minimal overhead to the Byzantine commit subprotocol, so the performance cost is worth the added benefit of being able to detect and remove faulty.

8.2 Fault detection in permissionless BFT systems.

Permissionless BFT systems, or blockchain systems, are designed to prioritize high availability in exchange for weak consistency, and as such, they are prone to a number of attacks that can cause inconsistencies between replicas and the rest of the network. We explored several ways in which we can enhance fault detection in blockchain systems and presented novel mechanisms to detect two major consistency attacks: `Eclipse Attacks` and `Execution Fork Attacks`. Additionally, we looked into how we can enhance fault detection on-chain by utilizing the rich features provided by smart contracts to encode fault detection mechanisms.

First, in detecting `Eclipse Attacks`, we realize that current research into detecting these types of attacks requires modifications to the communication layer to limit or reset peer connections or induce a high cost by requiring the user to constantly send transactions to the chain to verify that they are receiving data from the main chain. We presented a novel eclipse attack detection mechanism that involves monitoring the difficulty values of new blocks to determine if the system is under an eclipse attack. Our evaluation of the difficulty monitor shows that it is able to detect eclipse attacks with a negligible possibility of a false negative and a small number of false positives.

Second, we looked into enhancing fault detection to detect `Execution Fork Attacks` in state channels. Prior works that address this type of attack did not consider the difficulty of sending transactions during network congestion and only focused on ensuring that the counterparty (or a delegated third party) can see the attack happening. We presented a novel construction of payment channel contracts that takes this difficulty into account and allows the

counterparty to detect execution fork attacks and dispute them on-chain even when the network is congested.

Lastly, we looked into enhancing on-chain fault detection in blockchain systems in two ways. We first presented a smart contract framework that is able to authenticate digitally signed messages from off-chain replicas. This allows us to manage membership on-chain and replace replicas when they send equivocating messages, when their credentials are leaked, or when they are voted out by a quorum of replicas. We used this smart contract framework to build Decentagram, a novel decentralized publish/subscribe messaging system that is the first to provide on-chain authentication and on-chain notification. Then, we presented a smart contract that can be used as a heartbeat failure detector for decentralized applications in which two (or more) trusted parties need to encode service agreements in smart contracts. The incentivized heartbeat mechanism in our contract allows clients to detect whether the service provider is Byzantine and intentionally withholds responses to their requests and punish the service provider accordingly.

8.3 Future work.

There are several promising directions for future work in enhancing fault detection in both permissioned and permissionless BFT systems.

Reactive Reconfiguration in Resource-Efficient BFT. In practice, permissioned BFT protocols are generally less favored than CFT protocols due to the higher resource requirements (needing $3f + 1$ replicas instead of $2f + 1$ to tolerate f faults). One approach to reducing this overhead is through the use of trusted components [207, 63, 73, 67]. By using trusted com-

ponents to construct and validate consensus messages, Byzantine replicas cannot equivocate without being detected since they cannot tamper with the code running inside the trusted components. However, similar to other permissioned BFT protocols that we have seen so far, a successful detection of a Byzantine replica in these trusted BFT systems still leaves the faulty replica in the active replica set. This opens the possibility of integrating Phoenix within the trusted component to facilitate reactive reconfiguration to remove faulty replicas.

Applications for On-chain Fault Detection. As we have shown in this dissertation, the ability to automatically detect faults on-chain can be useful in constructing Pub/Sub and FaaS frameworks, two popular computing paradigms. Using smart contracts to manage membership and fault detection mechanisms to make changes to this membership can be useful in a wide array of other applications such as collaborative computing [221, 220, 148], resource discovery [144, 161, 177], and gossip-based protocols [10, 92, 106].

Other Consistency Attacks in Blockchains. Due to the inherently weak consistency guarantee in blockchains, `Eclipse Attacks` and `Execution Fork Attacks` are not the only types of consistency attacks that exist in these systems. For instance, in PoS blockchains, attacks such as Bribery Attacks [35], Stake-bleeding attacks [93], and Balance Attacks [164] can be used by an adversary to cause revert blocks. Additionally, other blockchains like `Proof-of-Contribution` [217] and `Proof-of-Storage` [109] each have a different underlying consensus algorithm that is used to produce blocks and are thus vulnerable to consistency attacks in different ways. We believe that exploring fault detection in blockchains other than PoW and PoS and how they can be used to prevent consistency attacks as a promising area of

research.

Appendix A

Correctness of Phoenix

In this Appendix, we prove the correctness properties of Phoenix. First, we will prove that reactive reconfiguration protocol is safe, i.e., that consensus decisions are not lost across reconfigurations. Then, we will prove that reactive reconfiguration is live, i.e., when the system becomes stuck from not having enough replicas, it will be able to reach a configuration where there are enough replicas to once again make progress.

A.1 Phoenix

Lemma 1. *Two commit quorums intersect in at least one replica, and this replica is not Byzantine faulty.*

Proof. Let CQ1 and CQ2 be two distinct commit quorums. Because each commit quorum is of size $n - f_B$, CQ1 and CQ2 must intersect in at least $f_B + f_C + 1$ replicas. Out of these $f_B + f_C + 1$ replicas, at least one replica is not Byzantine faulty nor crash faulty. □

Lemma 2. *A configuration manager concluding a voting round means that it has received one vote containing the latest consensus decision.*

Proof. Suppose that the latest consensus decision d is for consensus instance i containing a request r from client c, and that client c has received enough replies to consider r to be completed. Since the client considers r completed, any reconfiguration that is triggered by a successful voting round must preserve d, i.e., correct replicas in the next configuration must have d in their decision log. A request only completes if a client has received enough replies, therefore c must have received $n - f_B$ replies for r, and since a replica only replies to the client only after it has decided on d, this means that $n - f_B$ replicas have decided on d and have it in their decision log. Let RQ be the reply quorum containing the $n - f_B$ replies from replicas that have decided on d.

In Phoenix, a CM concludes a voting round when it has received vote quorum VQ of $n - f_B - f_C$ matching votes from distinct replicas. RQ intersects VQ in at least $f_B + 1$ replicas, with at least one of these replicas being non-Byzantine. Although the Byzantine replicas can truncate their decision logs to exclude d and send in some other consensus decision that precedes d, at least one non-Byzantine replica will have sent in a vote containing d. The vote message containing d from this replica will be chosen because d contains signed *accept* messages from $n - f_B$ distinct replicas that have committed r. □

Lemma 3. *If a client considers a request completed, then a reconfiguration will result in a configuration where at least $n - f_B$ replicas have the latest consensus decision in their decision log.*

Proof. Suppose again that the latest consensus decision d is for consensus instance i containing a request r from client c, and that client c has received enough replies to consider r to be completed. Let $\pi = (S)$ be the system with active replica set S before reconfiguration and $\pi' = (S')$ be the system with active replica set S' after reconfiguration,and let $D \subseteq S$ and $D' \subseteq S'$ be the sets that contains the consensus decision d within S and S', respectively. Also, let replica a be the replica in S that is being replaced by replica b in S'. We will consider two cases.

(1) $a \in D$**.** In this case, a has d in its decision log. Then, $D' = D - \{a\}$ and $|D'| = |D| - 1 \geq n - f_B - 1$, since $|D| \geq n - f_B$ (client considered the r completed, so d must be in at least $n - f_B$ replica's decision logs). By Lemma 2, a successful reconfiguration means that replica b has received the *join* message from the CM containing d, and will perform a state request so that its decision log matches the decision log with d as the latest consensus decision. Therefore, b in π' will also have d, and so we have $|D'| \geq n - f_B - \{a\} + \{b\} = n - f_B$.

(2) $a \notin D$**.** In this case, a does not have d in its decision log. Then, $D' = D$ and $|D'| = |D| \geq n - f_B$. Again, by Lemma 2, a successful reconfiguration means that replica b has d in its decision log, and so $|D'| \geq n - f_B - \{a\} + \{b\} = n - f_B + 1$.

In both cases, we have $|D'| \geq n - f_B$, so at least $n - f_B$ replicas in π' have d in their decision log. □

Theorem 4. *Reactive reconfiguration in Phoenix preserves safety.*

Proof. Let $\pi = (S)$ be the system with active replica set S before reconfiguration and $\pi' = (S')$ be the system with active replica set S' after reconfiguration,and let $D \subseteq S$ be the

set that contains the consensus decision d within S. By Lemma 3, we know that the set $D' \subseteq S'$ containing the replicas with d is of size $|D'| \geq n - f_B$. Suppose that after reconfiguration, a client c completes a request r' that was proposed in consensus instance i' and resulting in consensus decision d'. This means that d' is the latest consensus decision and it is in a slot succeeding d. Suppose for contradiction that d' is in the same slot as d, i.e., overwriting d. But, c completing r' means that c has received $n - f_B$ replies for r', so $n - f_B$ replicas have decided on d' and have it in their decision log. By Lemma 1, this set intersects with D' in at least one correct replica who have decided both d and d'. This is impossible, because the Byzantine commit algorithm forbids a correct replica from deciding on two different values in the same slot. □

Theorem 5. *Reactive reconfiguration in Phoenix is live.*

Proof. Let $\pi = (S^0)$ be the initial system with active replica set S^0, and suppose that after some period of time, the system has the configuration $\pi^j = (S^j)$ and undergoes a reconfiguration to $\pi^k = (S^k)$, with $k = j + 1$. We define $f^j = f_B^j + f_C^j$ to be the total number of faulty replicas in S^j and $f^k = f_B^k + f_C^k$ to be the total number of faulty replicas in S^k, and let $C^j = n^j - f^j$ and $C^k = n^k - f^k$ be the non-faulty replicas that can vote to reconfigure in S^j and S^k, respectively. Also, let H^k be the replicas in S^k that are able to send consensus messages in π^k. Suppose that $a \in S^j$ is the replica being replaced by replica b. Now, we consider two cases for a.

(1) $a \in f^j$**.** In this case, a is a faulty replica in S^j, so replacing it will leave us with $C^k = C^j + \{b\}$ non-faulty replicas in S^k that can vote to reconfigure. Since $C^k = C^j + 1 \geq n^k - f^k$, we have at least $n^k - f^k$ non-faulty replicas that can vote in π^k.

(2) $a \notin f^j$**.** In this case, a is a non-faulty replica in S^j, so replacing it will leave us with $C^k = C^j - \{a\} + \{b\}$ non-faulty replicas in S^k that can vote to reconfigure. Since $n^j = n^k$ and $C^j = C^k = n^k - f^k$, we have enough non-faulty replicas in π^k that can vote to reconfigure.

After reconfiguration, if $|H^k| \geq n^k - f_B^k$, then the system will be able to resume processing client requests. However, suppose that $|f^j| > 1$ and that $|H^k| < n^k - f_B^k$, i.e., the number of simultaneous faults is large enough that there are insufficient number of replicas to commit new requests. In this case, we can repeat the reconfiguration process, with each successful reconfiguration falling into one of the two cases above. Eventually, we will reach a configuration $g > k$ with a configuration π^g where $|H^g| \geq n^g - f_B^g$.

□

A.2 Sync Phoenix

Theorem 6. *Reactive reconfiguration in Sync Phoenix preserves safety.*

Proof. Suppose that the current configuration is $\pi^i = (S^i)$ and that the latest consensus decision d is for consensus instance i containing a request r from client c, and that client c has received enough replies to consider r completed. Let L_d be the longest valid decision log that contains d, i.e., L_d has no gaps in the log, and each of the consensus decisions in L_d is valid, with d being the latest consensus decision. Since c considers r completed, we have $D^i \subseteq S^i$ with L_d. Let t_s be the time at which the CM receives $f_B^i + 1$ matching votes to conclude a voting round, sends out the *vote-request* messages, and starts the timer. At time $t_s + \Delta$, where Δ is the

maximum network delay, these *vote-request* messages will have reached all the replicas in S^i. Given that $D = n - f_B = 3f_B + 1 - f_B = 2f_B + 1$ replicas with L_d, at least one non-faulty replica that receives the *vote-request* message will have L_d. Therefore, at time $t_s + 2\Delta$ when the CM expects to receive all the remaining votes, it will have received at least one vote containing L_d. Because reconfiguration is strictly synchronous, it doesn't matter if the replica being replaced is in D, since after a new replica joins (and receives L_d from the CM) at time $t_s + 3\Delta$, at least $n - f_B$ replicas will have L_d in the new configuration. □

Theorem 7. *Reactive reconfiguration in Sync Phoenix is live.*

Proof. In Sync Phoenix, we assume that the underlying network is synchronous, so the message delivery between replicas is bounded by some maximum network delay Δ. The voting protocol in Phoenix forbids correct replicas from voting against themselves, and in Sync Phoenix correct replicas will also not vote against other correct replicas because replicas that send late messages are faulty. Additionally, Byzantine replicas cannot collude to successfully vote out another replica because they can account for at most f_B votes, therefore a reconfiguration can only be triggered to vote out a replica that is either in f_B or f_C. Similar to the proof for Theorem 5, we will consider two possible outcomes. If a reconfiguration results in a system $\pi' = S'$ with $H' \geq n' - f'_B$, then the system will be able to resume processing client requests. However, if $|f_B| > 1$ and $H' < n' - f'_B$, then we can repeat the reconfiguration process, replacing replicas in f_B or f_C until we reach some configuration $\pi^g = (S^g)$ where $H^g \geq n^g - f^g_B$. □

Bibliography

[1] Bitcoin cash. https://bitcoincash.org/, 2021 (accessed December 10, 2021).

[2] Ethereum average gas price chart. https://etherscan.io/chart/gasprice, 2021 (accessed December 10, 2021).

[3] Private altruist watchtowers. https://github.com/lightningnetwork/lnd/blob/master/docs/watchtower.md, 2021 (accessed December 10, 2021).

[4] Raiden monitoring service. https://raiden-network-specification.readthedocs.io/en/latest/monitoring_service.html, 2021 (accessed December 10, 2021).

[5] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.

[6] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma. *arXiv preprint arXiv:1801.10022*, 2018.

[7] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[8] Bithin Alangot, Daniel Reijsbergen, Sarad Venugopalan, and Pawel Szalachowski. Decentralized lightweight detection of eclipse attacks on bitcoin clients. In *2020 IEEE international conference on Blockchain (Blockchain)*, pages 337–342. IEEE, 2020.

[9] Maher Alharby and Aad Van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*, 2017.

[10] André Allavena, Alan Demers, and John E Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, 2005.

[11] Amazon Web Services. Amazon SNS, 2023.

[12] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE, 2019.

[13] Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. Seemore: A fault-tolerant protocol for hybrid cloud environments. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1345–1356. IEEE, 2020.

[14] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[15] Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio, and Nikolaus Thümmel. PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, SRDS '18, pages 123–132, New York, NY, USA, October 2018. Institute of Electrical and Electronics Engineers.

[16] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezbft: Decentralizing byzantine fault-tolerant state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 565–577. IEEE, 2019.

[17] Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, et al. Decentralized and stateful serverless computing on the internet computer blockchain. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 329–343, 2023.

[18] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, pages 164–186. Springer, 2017.

[19] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):1–45, 2015.

[20] Georgia Avarikioti, Eleftherios Kokoris Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous payment channels. *arXiv preprint arXiv:1905.11360*, 2019.

[21] AWS. Aws s3 availability event. https://status.aws.amazon.com/s3-20080720.html. Accessed: 2021-09-20.

[22] Jean Bacon, David M. Eyers, Jatinder Singh, and Peter R. Pietzuch. Access control in publish/subscribe systems. In *Proceedings of the Second International Conference on Distributed Event-Based Systems*, DEBS '08, pages 23–34, New York, NY, USA, July 2008. Association for Computing Machinery.

[23] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, 2014.

[24] Kyle Banker, Douglas Garrett, Peter Bakkum, and Shaun Verch. *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster, 2016.

[25] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

[26] Beaconscan. Statistics & charts, mainnet beacon chain (phase 0) ethereum 2.0 explorer. `https://beaconscan.com/statistics`, 2023.

[27] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1521–1538, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.

[29] Christian Berger, Sadok Ben Toumia, and Hans P Reiser. Does my bft protocol implementation scale? In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, pages 19–24, 2022.

[30] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From byzantine replication to blockchain: Consensus is only the beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 424–436. IEEE, 2020.

[31] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[32] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of ethereum: An initial quantitative analysis. In *2019 ieee international conference on service-oriented system engineering (sose)*, pages 167–176. IEEE, 2019.

[33] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, December 1985. Association for Computing Machinery.

[34] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

[35] Joseph Bonneau. Why buy when you can rent? bribery attacks on bitcoin-style consensus. In *International Conference on Financial Cryptography and Data Security*, pages 19–26. Springer, 2016.

[36] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of byzantine state-machine replication. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[37] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Ari Juels, and Andrew Miller. Chainlink off-chain reporting protocol. Technical report, Chainlink Labs, February 2021.

[38] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343–477. Portland, OR, 2000.

[39] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016.

[40] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[41] Vitalik Buterin. A next generation smart contract & decentralized application platform. Technical report, Ethereum Foundation, 2013. Ethereum White Paper.

[42] Vitalik Buterin. Hard fork completed. `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`, 2016.

[43] Vitalik Buterin. An incomplete guide to rollups. `https://vitalik.ca/general/2021/01/05/rollup.html`, 2021 (accessed December 10, 2021).

[44] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[45] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[46] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[47] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[48] Miguel Castro. Practical Byzantine Fault Tolerance. `https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf`, 2001. [Online; accessed 30-March-2023].

[49] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314, 2002.

[50] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, 2000.

[51] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance.

[52] Chainalysis Inc. Chainalysis oracle for sanctions screening, 2023.

[53] TH Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, 2018.

[54] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[55] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[56] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[57] Badrish Chandramouli and Jun Yang. End-to-end support for joins in large-scale publish/subscribe systems. *Proceedings of the VLDB Endowment*, 1(1):434–450, August 2008.

[58] Anamika Chauhan, Om Prakash Malviya, Madhav Verma, and Tejinder Singh Mor. Blockchain and scalability. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 122–128. IEEE, 2018.

[59] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys*, 53(3), June 2020.

[60] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, 51(5):561–580, 2002.

[61] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[62] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[63] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

[64] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[65] Chameleon Cloud. *A configurable experimental environment for large-scale edge to cloud research*, 2021 (accessed September 2, 2021).

[66] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

[67] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 174–183. IEEE, 2004.

[68] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[69] Chris Dannen. *Introducing Ethereum and solidity*, volume 318. Springer, 2017.

[70] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[71] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 1–16, 2022.

[72] Omar Dib, Kei-Leo Brousmiche, Antoine Durand, Eric Thea, and Elyes Ben Hamida. Consortium blockchains: Overview, applications and challenges. *Int. J. Adv. Telecommun*, 11(1):51–64, 2018.

[73] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-efficient byzantine fault tolerance. *IEEE transactions on computers*, 65(9):2807–2819, 2015.

[74] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In *European Dependable Computing Conference*, pages 71–87. Springer, 1999.

[75] Sisi Duan, Chao Liu, Xin Wang, Yusen Wu, Shuai Xu, Yelena Yesha, and Haibin Zhang. Intrusion-tolerant and confidentiality-preserving publish/subscribe messaging. In *2020*

International Symposium on Reliable Distributed Systems (SRDS), pages 319–328. Institute of Electrical and Electronics Engineers, September 2020.

[76] Sisi Duan, Sean Peisert, and Karl N Levitt. hbft: speculative byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, 12(1):58–70, 2014.

[77] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[78] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 106–123. IEEE, 2019.

[79] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 801–815, 2019.

[80] Ayman Esmat, Martijn de Vos, Yashar Ghiassi-Farrokhfal, Peter Palensky, and Dick Epema. A novel decentralized platform for peer-to-peer energy trading market with blockchain technology. *Applied Energy*, 282:116123, 2021.

[81] Ethereum Foundation. Go ethereum - official go implementation of the ethereum protocol, 2023.

[82] Ethereum Team. Go ethereum: Official go implementation of the ethereum protocol. `https://geth.ethereum.org/`.

[83] Ethereum Team. Release let there be light (v1.5.0) ethereum/go-ethereum. `https://github.com/ethereum/go-ethereum/releases/tag/v1.5.0`, 2016.

[84] Ethereum Team. Byzantium hf announcement. `https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/`, 2017.

[85] Ethereum Team. Release version 0.4.14 ethereum/solidity. `https://github.com/ethereum/solidity/releases/tag/v0.4.14`, 2017.

[86] Ethereum Team. Release version 0.6.2 ethereum/solidity, 2020.

[87] Etherscan. Forked blocks. `https://etherscan.io/blocks_forked/`, 2023.

[88] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[89] Benjamin Eze, Craig Kuziemsky, Liam Peyton, Grant Middleton, and Alain Mouttham. Policy-based data integration for e-health monitoring processes in a B2B environment: Experiences from canada. *Journal of Theoretical and Applied Electronic Commerce Research*, 5(1):56–70, April 2010.

[90] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[91] Roy Friedman and Robbert Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*, pages 233–242. IEEE, 1997.

[92] Ayalvadi J Ganesh, A-M Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE transactions on computers*, 52(2):139–149, 2003.

[93] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *2018 Crypto Valley conference on Blockchain technology (CVCBT)*, pages 85–92. IEEE, 2018.

[94] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016.

[95] Sara Ghaemi, Hamzeh Khazaei, and Petr Musilek. Chainfaas: An open blockchain-based serverless platform. *IEEE Access*, 8:131760–131778, 2020.

[96] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[97] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[98] go-ethereum. consensus.go. `https://github.com/ethereum/go-ethereum/blob/4b2ff1457ac28fb2894485194e0e344e84c2bcd7/consensus/ethash/consensus.go`, 2020.

[99] Muhammed Golec, Deepraj Chowdhury, Shivam Jaglan, Sukhpal Singh Gill, and Steve Uhlig. Aiblock: Blockchain based lightweight framework for serverless computing using ai. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 886–892. IEEE, 2022.

[100] Muhammed Golec, Sukhpal Singh Gill, Mustafa Golec, Minxian Xu, Soumya K Ghosh, Salil S Kanhere, Omer Rana, and Steve Uhlig. Blockfaas: Blockchain-enabled serverless computing framework for ai-driven iot healthcare applications. *Journal of Grid Computing*, 21(4):63, 2023.

[101] Google Cloud. What is Pub/Sub?, May 2023.

[102] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[103] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Fault-tolerant distributed transactions on blockchain. *Synthesis Lectures on Data Management*, 16(1):1–268, 2021.

[104] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160*, 2020.

[105] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 754–764. IEEE, 2020.

[106] Zygmunt J Haas, Joseph Y Halpern, and Li Li. Gossip-based ad hoc routing. *IEEE/ACM Transactions on networking*, 14(3):479–491, 2006.

[107] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for byzantine fault detection. In *HotDep*, 2006.

[108] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. Technical report, Sun Microsystems, Santa Clara, CA, December 2002.

[109] Kun He, Jing Chen, Ruiying Du, Qianhong Wu, Guoliang Xue, and Xiang Zhang. Deypos: Deduplicatable dynamic proof of storage for multi-user environments. *IEEE Transactions on Computers*, 65(12):3631–3645, 2016.

[110] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144, 2015.

[111] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[112] Jochen Hoenicke. Johoe's bitcoin mempool statistics. https://jochen-hoenicke.de/queue. Accessed: 2023-03-29.

[113] Heidi Howard, Aleksey Charapko, and Richard Mortier. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, pages 186–190, 2021.

[114] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[115] Kexin Hu, Kaiwen Guo, Qiang Tang, Zhenfeng Zhang, Hao Cheng, and Zhiyang Zhao. Leopard: Towards high throughput-preserving bft for large-scale systems. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 157–167. IEEE, 2022.

[116] Yennun Huang and Chandra Kintala. Software implemented fault tolerance: Technologies and experience. In *FTCS*, volume 23, pages 2–9. IEEE Computer Society Press, 1993.

[117] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[118] Infura. Infura, a consensys formation. https://www.infura.io/. Accessed: 2023-03-29.

[119] Hudson Jameson. Faq: Upcoming ethereum hard fork. `https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/`, 2016.

[120] Hudson Jameson. Hard fork no. 4: Spurious dragon. `https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/`, 2016.

[121] Hudson Jameson. Ethereum constantinople/st. petersburg upgrade announcement. `https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/`, 2019.

[122] Hudson Jameson. Ethereum istanbul upgrade announcement. `https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement/`, 2019.

[123] Hudson Jameson. Ethereum muir glacier upgrade announcement. `https://blog.ethereum.org/2019/12/23/ethereum-muir-glacier-upgrade-announcement/`, 2019.

[124] Linpeng Jia, Keyuan Wang, Xin Wang, Lei Yu, Zhongcheng Li, and Yi Sun. Themis: An equal, unpredictable, and scalable consensus for consortium blockchain. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 235–245. IEEE, 2022.

[125] Kaustubh R Joshi, Matti A Hiltunen, William H Sanders, and Richard D Schlichting. Automatic model-driven recovery in distributed systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 25–36. IEEE, 2005.

[126] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Reliable and highly available distributed publish/subscribe service. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 41–50. Institute of Electrical and Electronics Engineers, September 2009.

[127] Lucianna Kiffer, Dave Levin, and Alan Mislove. Stick a fork in it: Analyzing the ethereum network partition. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 94–100, 2017.

[128] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. The securering protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 317–326. IEEE, 1998.

[129] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[130] Hermann Kopetz. Real time and dependability concepts. *Distributed systems*, 1993.

[131] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[132] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11 of *NetDB'11*, pages 1–7, New York, NY, USA, June 2011. Association for Computing Machinery.

[133] Seda Kul, Süleyman Eken, and Ahmet Sayar. Distributed and collaborative real-time vehicle detection and classification over the video streams. *International Journal of Advanced Robotic Systems*, 14(4), July 2017.

[134] Seda Kul, Isabek Tashiev, Ali Şentaş, and Ahmet Sayar. Event-based microservices with apache kafka streams: A real-time vehicle detection system based on type, color, and speed attributes. *IEEE Access*, 9:83137–83148, June 2021.

[135] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, and Yongdae Kim. Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash? In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 935–951. IEEE, 2019.

[136] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[137] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[138] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[139] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[140] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[141] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[142] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[143] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[144] Daniel Lazaro, Joan Manuel Marques, Josep Jorba, and Xavier Vilajosana. Decentralized resource discovery mechanisms for distributed computing in peer-to-peer environments. *ACM Computing Surveys (CSUR)*, 45(4):1–40, 2013.

[145] Peng Li, Toshiaki Miyazaki, and Wanlei Zhou. Secure balance planning of off-blockchain payment channel networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1728–1737. IEEE, 2020.

[146] Wenyu Li, Chenglin Feng, Lei Zhang, Hao Xu, Bin Cao, and Muhammad Ali Imran. A scalable multi-layer pbft consensus for blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1146–1160, 2020.

[147] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 485–500, 2016.

[148] Jing Long, Tong Chen, Quoc Viet Hung Nguyen, and Hongzhi Yin. Decentralized collaborative learning framework for next poi recommendation. *ACM Transactions on Information Systems*, 41(3):1–25, 2023.

[149] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[150] Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis, and Jayavel Shanmugasundaram. Scalable ranked publish/subscribe. *Proceedings of the VLDB Endowment*, 1(1):451–462, August 2008.

[151] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings 10th Computer Security Foundations Workshop*, pages 116–124. IEEE, 1997.

[152] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[153] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *Cryptology ePrint Archive*, 2018.

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

[155] Stephanos Matsumoto and Raphael M Reischuk. Ikp: Turning a pki around with decentralized automated incentives. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 410–426. IEEE, 2017.

[156] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 16–30, 2019.

[157] Ralph C Merkle. A certified digital signature. In *Advances in cryptology—CRYPTO'89 proceedings*, pages 218–238. Springer, 2001.

[158] Andrew Miller. Permissioned and permissionless blockchains. *Blockchain for distributed systems security*, pages 193–204, 2019.

[159] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *International Conference on Financial Cryptography and Data Security*, pages 508–526. Springer, 2019.

[160] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[161] Ilir Murturi and Schahram Dustdar. A decentralized approach for resource discovery using metadata replication in edge networks. *IEEE Transactions on Services Computing*, 15(5):2526–2537, 2021.

[162] Farid Nait-Abdesselam, Brahim Bensaou, and Tarik Taleb. Detecting and avoiding wormhole attacks in wireless ad hoc networks. *IEEE Communications Magazine*, 46(4):127–133, 2008.

[163] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

[164] Christopher Natoli and Vincent Gramoli. The balance attack or why forkable blockchains are ill-suited for consortium. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 579–590. IEEE, 2017.

[165] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[166] Michael Neuder, Daniel J Moroz, Rithvik Rao, and David C Parkes. Low-cost attacks on ethereum 2.0 by sub-1/3 stakeholders. *arXiv preprint arXiv:2102.02247*, 2021.

[167] Cong T Nguyen, Dinh Thai Hoang, Diep N Nguyen, Dusit Niyato, Huynh Tuong Nguyen, and Eryk Dutkiewicz. Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities. *IEEE access*, 7:85727–85745, 2019.

[168] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[169] A Pinar Ozisik, Gavin Andresen, George Bissias, Amir Houmansadr, and Brian Levine. Graphene: A new protocol for block propagation using set reconciliation. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 420–428. Springer, 2017.

[170] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical report, Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[171] Lauri I. W. Pesonen and Jean Bacon. Secure event types in content-based, multi-domain publish/subscribe systems. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM '05, pages 98–105, New York, NY, USA, September 2005. Association for Computing Machinery.

[172] PoolWatch.io. Best ethereum mining pools for 2020, 2020.

[173] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[174] Bernd Prünster, Alexander Marsalek, and Thomas Zefferer. Total eclipse of the heart–disrupting the {InterPlanetary} file system. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3735–3752, 2022.

[175] RabbitMQ. RabbitMQ, 2023.

[176] Gowri Sankar Ramachandran, Kwame-Lante Wright, Licheng Zheng, Pavas Navaney, Muhammad Naveed, Bhaskar Krishnamachari, and Jagjit Dhaliwal. Trinity: A byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 227–235. Institute of Electrical and Electronics Engineers, May 2019.

[177] Rajiv Ranjan, Aaron Harwood, and Rajkumar Buyya. Peer-to-peer-based resource discovery in global grids: a tutorial. *IEEE Communications Surveys & Tutorials*, 10(2):6–33, 2008.

[178] Benjamin Reed. Gsocfailuredetector. `https://cwiki.apache.org/confluence/display/ZOOKEEPER/GSoCFailureDetector/`, 2011.

[179] Hans P Reiser and Rudiger Kapitza. Hypervisor-based efficient proactive recovery. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 83–92. IEEE, 2007.

[180] Michael K Reiter. The rampart toolkit for building high-integrity services. In *Theory and practice in distributed systems*, pages 99–110. Springer, 1995.

[181] Michael K Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[182] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[183] Rodrigo Rodrigues and Barbara Liskov. Byzantine fault tolerance in long-lived systems. 2004.

[184] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. Ccf: A framework for building confidential verifiable replicated services. *Technical report, Microsoft Research and Microsoft Azure*, 2019.

[185] Ryan Schneider. Bloom filter false positive rate w/ ERC-20/721, 2018.

[186] Kenji Saito and Hiroyuki Yamada. What's so different about blockchain?—blockchain is a probabilistic state machine. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 168–175. IEEE, 2016.

[187] Fahad Saleh. Blockchain without waste: Proof-of-stake. *The Review of financial studies*, 34(3):1156–1190, 2021.

[188] Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. *ICDCN*, 12:153–167, 2012.

[189] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[190] Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 353–362. IEEE, 2010.

[191] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, et al. {IA-CCF}: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 467–491, 2022.

[192] Atul Singh, Tsuen-Wan Ngan, Peter Druschel, Dan S Wallach, et al. Eclipse attacks on overlay networks: Threats and defenses. 2006.

[193] Corwin Smith, Sebastian Supreme, Chirag Badhe, and Tyler Pfledderer. Ethereum proof-of-stake attack and defense. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attack-and-defense/, June 2023.

[194] Joao Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[195] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2009.

[196] Jutta Steiner. Security alert [consensus issue]. https://blog.ethereum.org/2015/08/20/security-alert-consensus-issue/, 2015.

[197] Jutta Steiner. Security alert – [implementation bug in go clients causing increase in difficulty – fixed – miners check and update go clients]. https://blog.ethereum.org/2015/09/03/security-alert-implementation-bug-in-go-clients-causing-increase-in-difficulty-fixed-miners-check-and-update-go-clients-if-necessary/, 2015.

[198] Paul Stelling, Cheryl DeMatteis, Ian Foster, Carl Kesselman, Craig Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2:117–128, 1999.

[199] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[200] Tuan Tran, Faisal Nawab, Peter Alvaro, and Owen Arden. Unstick yourself: Recoverable byzantine fault tolerant services. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2023.

[201] Tuan Tran and Haofan Zheng. payment-channel-congestion, 2021.

[202] Tuan Tran, Haofan Zheng, Peter Alvaro, and Owen Arden. Payment channels under network congestion. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5. IEEE, 2022.

[203] Truffle Suite. What is ganache?, 2023.

[204] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[205] Stephan Tual. Ethereum protocol update 1. https://blog.ethereum.org/2015/08/04/ethereum-protocol-update-1/, 2015.

[206] Uniswap Labs. Uniswap protocol, 2023.

[207] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[208] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. IEEE, 2011.

[209] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 95–112, 2019.

[210] Peipei Wang, Daniel J Dean, and Xiaohui Gu. Understanding real world data corruptions in cloud systems. In *2015 IEEE International Conference on Cloud Engineering*, pages 116–125. IEEE, 2015.

[211] Jeffrey Wilcke. The ethereum network is currently undergoing a dos attack. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/, 2016.

[212] Jeffrey Wilcke. Homestead release. https://blog.ethereum.org/2016/02/29/homestead-release/, 2016.

[213] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[214] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Foundation, 2022.

[215] Kwame-Lante Wright, Martin Martinez, Uday Chadha, and Bhaskar Krishnamachari. Smartedge: A smart contract for edge computing. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1685–1690. Institute of Electrical and Electronics Engineers, July 2018.

[216] Guangquan Xu, Bingjiang Guo, Chunhua Su, Xi Zheng, Kaitai Liang, Duncan S Wong, and Hao Wang. Am i eclipsed? a smart detector of eclipse attacks for ethereum. *Computers & Security*, 88:101604, 2020.

[217] Tengfei Xue, Yuyu Yuan, Zahir Ahmed, Krishna Moniz, Ganyuan Cao, and Cong Wang. Proof of contribution: A modification of proof of work to increase mining efficiency. In

2018 IEEE 42nd annual computer software and applications conference (COMPSAC), volume 1, pages 636–644. IEEE, 2018.

[218] YCHARTS. Ethereum average gas limit, 2023.

[219] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[220] Liang Yuan, Qiang He, Feifei Chen, Jun Zhang, Lianyong Qi, Xiaolong Xu, Yang Xiang, and Yun Yang. Csedge: Enabling collaborative edge storage for multi-access edge computing based on blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1873–1887, 2021.

[221] Liang Yuan, Qiang He, Siyu Tan, Bo Li, Jiangshan Yu, Feifei Chen, Hai Jin, and Yun Yang. Coopedge: A decentralized blockchain-based platform for cooperative edge computing. In *Proceedings of the Web Conference 2021*, pages 2245–2257, 2021.

[222] Yuanyuan Zhao and Daniel C. Sturman. Dynamic access control in a content-based publish/subscribe system with delivery guarantees. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 60–60. Institute of Electrical and Electronics Engineers, July 2006.

[223] Haofan Zheng and Tuan Tran. Decentdiffmonitorexpr, 2020.

[224] Haofan Zheng and Tuan Tran. Gethdbreader, 2020.

[225] Haofan Zheng, Tuan Tran, and Owen Arden. Total eclipse of the enclave: detecting eclipse attacks from inside tees. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5. IEEE, 2021.

[226] Nejc Zupan, Kaiwen Zhang, and Hans-Arno Jacobsen. Hyperpubsub: A decentralized, permissioned, publish/subscribe service using blockchains: Demo. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, Middleware '17, pages 15–16, New York, NY, USA, 2017. Association for Computing Machinery.