Revamping Byzantine Fault Tolerant State Machine Replication with Decentralization, Trusted Execution, and Practical Transformations

Balaji Arun

Preliminary Exam

Doctor of Philosophy
in
Computer Engineering

May 9, 2019
Blacksburg, Virginia

Keywords: Distributed Systems, Fault Tolerance, Trusted Execution Environments, Geographical Scalability, Byzantine Fault Tolerance, Hybrid Fault Tolerance

Revamping Byzantine Fault Tolerant State Machine Replication with Decentralization, Trusted Execution, and Practical Transformations

Balaji Arun

(ABSTRACT)

Cloud computing's ubiquitous infrastructure-as-a-service (IaaS) model has enabled a broad range of enterprise organizations to roll out online services at low cost. Cloud's multi-region support allows computational resources to be instantiated from different data centers around the world, enabling new classes of geographical-scale (or "geo-scale") applications. State machine replication (SMR) is the de facto standard for building highly scalable and available distributed applications and services. SMR replicates a service across a set of nodes, and executes client operations on the replicas in an agreed-upon total order, ensuring consistency of the replicated state. The problem of determining a total order reduces to one of *consensus*.

State-of-the-art consensus protocols are inadequate for newer classes of applications such as blockchains and for geo-scale infrastructures. The widely used Crash Fault Tolerance (CFT) fault model of consensus protocols is prone to malicious and adversarial behaviors and non-crash faults such as software bugs. Byzantine fault-tolerant models permit stronger failure adversaries. However, state-of-the-art Byzantine consensus protocols do not scale for geo-scale systems: they designate a primary replica for proposing total-orders, which becomes a bottleneck, and which yields sub-optimal latencies for far-away clients. Additionally, they do not exploit workload characteristics: commuting requests are also total-ordered, preventing their parallel execution.

To overcome these limitations and develop highly scalable and practical Byzantine SMR, this dissertation proposes: 1) ezBFT: *Fast, Leaderless Byzantine Fault-Tolerant (BFT) Consensus.* 2) Dester: *Fast, Leaderless Hybrid Fault-Tolerant Consensus.* 3) Bumblebee: *Methodology to transform CFT protocols to BFT using Trusted Hardware.*

ezBFT is a novel leaderless, distributed consensus protocol capable of tolerating byzantine faults. ezBFT's main goal is to minimize the client-side latency in WAN deployments. It achieves this by (i) having no designated primary replica, and instead, enabling every replica to order the requests that it receives from clients; (ii) using only three communication steps to order requests in the common case; and (iii) involving clients actively in the consensus process. Our experimental evaluation reveals that ezBFT improves client-side latency by as much as 40% over state-of-the-art byzantine fault-tolerant protocols including PBFT, FaB, and Zyzzyva.

Dester is a leaderless hybrid state machine replication protocol that incorporates a novel trusted subsystem, called TruDep, for achieving high performance in geo-scale deployments. Dester allows any replica to propose and commit client commands in two communication steps in most practical situations, while clients minimize latency by sending commands to the closest replica. Through evaluation, we show that Dester is able to reduce latency by as much as 30% compared to recent state-of-the-art solutions, while providing better throughput and tolerance to faults.

Bumblebee is a general methodology for transforming CFT protocols to tolerate byzantine faults by incorporating a trusted subsystem realized using trusted execution environments. This is because distributed systems that employ crash fault-tolerant (CFT) consensus-based state machine replication are incapable of handling non-crash faults that are increasingly common. We demonstrate the feasibility of the Bumblebee approach by transforming common CFT protocols including Raft, Paxos, and WPaxos into their hybrid counterparts. By incorporating our approach in etcd's Raft consensus, we show that overheads due to hybridization are less than 30% in terms of throughput.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

The last decade has witnessed an ever increasing proliferation of online services enabled by ubiquitous cloud platforms. Cloud providers charge for per unit of computation resource utilized, thus minimizing the total cost of ownership of cloud resources. This has enabled businesses ranging from startups to large-scale enterprises to take advantage of cloud's flexibility to develop highly available (i.e., round-the-clock) applications and services. Moreover, the ubiquity of multi-regional cloud allows computational resources to be instantiated at different data centers around the world, enabling new classes of geographical-scale ("geo-scale") applications [3, 7].

1.1.1 Replication

Historically, developers have relied on *replication* to build highly scalable and available services, because a single host cannot handle the many millions of client requests issued per second. Moreover, distributed systems, in general, are prone to faults such as machine crashes, network outages, and software bugs. Replication has been perhaps the most viable solution to increase scalability and availability of applications for many decades. Developers have used replication to build business- and mission-critical services such as databases [4, 35], key-value stores [5], and coordination systems [48].

Replication of stateful systems is challenging as the *replicas*, i.e., nodes that maintain a copy of the replicated state (e.g., key-value stores, tables, indexes), need to be kept consistent, which requires coordination among the replicas. The cost of coordination is exacerbated in geographically distributed services where replicas are spread across geographic locations around the world and are connected via high latency network links. Maintaining consistent copies allows stateful replication to be transparent to client applications. Thus, interfacing applications can be programmed to use the replicated system as if it were a non-replicated one – i.e., *one-copy* semantics [45] – which enables high programmability.

The state machine replication (SMR) approach has been widely used to build fault-tolerant replicated services [79]. In this approach, every replica in the system starts from the same initial state, execute client-issued operations (e.g., look-up, update operations in a key-value store) in the same order – a *total order* – across all replicas, and thereby reach the same final state. Consensus protocols can be used to build replicated state machines [79]. Simply put, consensus is the problem of selecting a value among a set of proposed values, which allows a total order to be determined. Paxos [60] is a well-known implementation of consensus that enables SMR. Implementations of consensus exist in many practical systems ranging from coordination services such as Zookeeper [48] to database systems such as CockroachDB [4].

1.1.2 Fault Models

Historically, many consensus protocols presented in the literature target two fault-tolerance models: crash fault-tolerance (CFT) and Byzantine fault-tolerance (BFT). The CFT model covers faults such as machine crashes and network outages, and requires a minimum of $N = 2f + 1$ replicas to tolerate a maximum of f failures. Example CFT consensus protocols include Paxos [60], EPaxos [72], Caesar [20], and M^2Paxos [78]. Many SMR systems that are in wide-spread production use today are based on the CFT model [3, 4, 5, 7, 35, 39, 48].

In contrast, the BFT model permits a stronger failure adversary. In addition to crash faults and network outages, this model covers malicious and adversarial behaviors, as well as non-crash faults such as hardware errors, corrupted data, software bugs, and data losses. BFT requires a minimum of $N = 3f + 1$ replicas to tolerate a maximum of f Byzantine failures. Example BFT consensus protocols include PBFT [30, 31], FaB [66], Zyzzyva [52], and Aliph [22].

Recently, BFT protocols have been used to solve consensus in a blockchain setting [25]. A blockchain is a *ledger*, a data structure that is replicated across a set of nodes, on which transactions (i.e., a sequence of operations) issued by clients are executed in an agreed-upon total order, which ensures consistency of the replicated (data structure) state. The problem of determining a total order reduces to one of consensus. For the class of *permissioned* ledgers, nodes are untrusted, so the problem further reduces to BFT consensus. [1] As a matter of fact, state machine replication is the most common way to implement permissioned ledgers [25].

The BFT model, in which the adversary is assumed to take control of faulty machines as well as the network in a coordinated way, is superfluous when such a strong adversary is unlikely. Precisely, for a class of distributed systems such as storage and database systems that are maintained by a single entity, non-crash faults typically include errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, bugs in software, hardware faults due to ever smaller circuits, and human mistakes that cause state corruptions and data losses. Therefore, for such systems, the hybrid fault-tolerance model [26] has been shown to yield greater resource savings while still tolerating Byzantine faults. This can particularly improve the reliability of databases and coordination

[1] In contrast, in permissionless ledgers, which seeks to avoid a central authority, any node can participate [25]. In permissioned ledgers, nodes are untrusted, but openness and anonymity are not goals [25].

systems, without the need for a special consistency verification process [68] or fault-recovery mechanisms [14]. Hybrid fault tolerant protocols employ a small trusted subsystem that can only fail by crashing to reduce the number of replicas required for *safe* replication from $3f+1$ to $2f+1$, matching the CFT requirement. Though some early hybrid solutions used ASICs and FPGAs resulting in high development complexity and very low performance [51, 61], the advent of trusted execution environments (TEEs) within commodity hardware such as Intel SGX [37] and ARM Trustzone [63] significantly reduces the performance overheads for hybrid protocols [26]. A software-based subsystem can be safe-guarded within these trusted environments providing the required level of trust.

1.1.3 Challenges

Existing state-of-the-art solutions have clear weaknesses that inhibit their ability to perform well in geo-scale deployments and provide strong consistency and security guarantees in events of non-crash failures.

Geo-scale deployments of BFT and Hybrid systems raise an important challenge: taming the high communication latency. Since replicas need to communicate with each other and with clients to reach consensus, the number of communication steps incurred directly impacts latency, as each step involves sending messages to potentially distant nodes. Moreover, state-of-the-art BFT and Hybrid protocols are primary-based: a replica in the system is bestowed the *primary* status and is responsible for assigning the total-order for executing the requests issued by clients. This can result in sub-optimal performance, especially in geo-scale settings. First, clients may originate from different regions around the world, but their requests must be sent to the primary even when other replica nodes are present much closer to the client than the primary. Thus, such clients will perceive sub-optimal latencies for processing their requests. Second, the primary must handle all client requests, irrespective of where they originate, thereby performing more work than other replicas. This causes load imbalance among the replicas, causing the primary to become a resource bottleneck, hampering scalability.

A *leaderless* protocol can solve the aforementioned problems. A client can send its requests to the geographically-closest replica and can continue to do so as long as that replica is *correct* (i.e., non-Byzantine). The replica can undertake the task of finding an order among all the concurrent requests in the system, execute the request on the shared state, and return the result. Since individual replicas share the workload as equal contributors in the system, performance scales. Leaderless consensus protocols [20, 73, 78] have been previously studied for the CFT model. However, to the best of our knowledge, no such investigations exist for the BFT model.

Furthermore, a plethora of distributed systems, in production use today, adopt crash fault-tolerant consensus protocols [3, 4, 7]. However, these systems can tolerate many of the fatal non-crash faults [36] by adopting a hybrid fault tolerance protocol. Today, this requires swapping out the existing CFT protocol implementation for a hybrid fault tolerant one. This causes unnecessary friction in software development as the new protocol would require rigorous testing and validation. If, instead, additional mechanisms are added to CFT protocols to tolerate byzantine failures, then development would become more incremental than a "ground-up" construction and thus would ease and facilitate adoption in practice. In doing so, the crux of CFT protocols should be maintained: requiring only $2f+1$ nodes to tolerate a maximum of f failures and reduced complexity compared to other fault models. In addition, the trusted subsystem must be minimal enough to easily ensure that their implementations are bug-free. This dissertation addresses these challenges.

1.2 Summary of Research Contributions

This dissertation underscores that state-of-the-art SMR solutions have clear weaknesses and limitations in geographical scale applications and services, and argues that leaderless consensus is key to achieving high performance and scalable SMR in these deployments. In that regard, this dissertation proposes two sets of techniques for building high performance, scalable, leaderless Byzantine Fault Tolerant consensus protocols, targeting blockchain and transactional systems.

Furthermore, this dissertation acknowledges that there are a plethora of distributed systems in production today adopting a crash fault-tolerant consensus solution, making them incapable of tolerating non-crash faults that are becoming increasingly common in many distributed systems today. For such systems, this dissertation proposes an approach to transition CFT protocols to tolerate Byzantine Faults with the help of trusted hardware.

To summarize, this dissertation makes the following contributions:

- ezBFT [19]. A leaderless protocol for solving BFT consensus quickly, i.e., in three communication steps, under low contention. This is done by exploiting commutativity of client requests and by exchanging request dependencies (Chapter 4).

- Dester [17]. A leaderless protocol for solving Hybrid consensus quickly, i.e., in three communication steps, under low contention. This is done by exploiting commutativity of client requests and by exchanging request dependencies. However, this approach is able to always maintain the size of quorums equal to the minimum that is necessary to solve hybrid consensus in an asynchronous system, i.e., the majority (Chapter 5).

- Bumblebee [18]: A general approach for transforming existing CFT protocols to tolerate Byzantine faults using trusted subsystems present in commodity hardware. This enables leveraging the investment in the CFT space for developing BFT protocols (Chapter 6).

1.2.1 ezBFT

First, this dissertation presents ezBFT, a leaderless Byzantine Fault Tolerant protocol that enables every replica in the system to process the requests received from the clients. Doing so (i) significantly reduces the client-side latency, (ii) distributes the load across replicas, and (iii) tolerates faults more effectively. Importantly, ezBFT delivers requests in *three* communication steps in normal operating conditions. To enable leaderless operation, ezBFT exploits a particular characteristic of client commands: *interference*. In the absence of concurrent interfering commands, ezBFT's clients receive a reply in an optimal three communication steps. When commands interfere, both clients and replicas coherently communicate to establish a consistent total-order, consuming an additional zero or two communication steps. ezBFT employs additional techniques such as client-side validation of replica messages and speculative execution to reduce communication steps in the common case, unlike CFT solutions [20, 73]. To understand how ezBFT fares against state-of-the-art BFT protocols, we implemented ezBFT and conducted an experimental evaluation using the AWS EC2 infrastructure, deploying the implementations in different sets of geographical regions. Our evaluation reveals that ezBFT improves client-side latency by as much as 40% over PBFT, FaB, and Zyzzyva.

1.2.2 Dester

Second, this dissertation proposes Dester, a leaderless hybrid fault-tolerant state machine replication protocol built *ground-up* for achieving high performance in the geo-scale environments. The leaderless nature allows every replica to process client commands by only relying on a set of closest replicas, thus providing low client-side latencies and high system throughput. Dester is, in part, made possible by a novel trusted subsystem called TruDep that provides the necessary trust required to ensure secure, leaderless operation. TruDep was designed specifically for trusted execution environments (such as Intel SGX) with a goal of minimizing the amount of trusted code, thus nullifying the possibility of bugs in the subsystem. We implemented Dester and evaluated it against state-of-the-art systems including Hybster and PBFT. For geo-scale deployment, we leverage SGX-capable virtual machine instances available in the Azure cloud platform [2]. Our experimental evaluations reveal that Dester provide up to 50% lower latency and 30% more throughput than Hybster in a geo-replicated setting.

1.2.3 Bumblebee

Finally, this dissertation proposes a general methodology for transforming CFT protocols to tolerate byzantine faults by incorporating a trusted subsystem realized using trusted execution environments. This dissertation identifies the fundamental design differences between

CFT, BFT, and Hybrid protocols, and develop key insights that are necessary for a CFT-to-BFT transformation. We develop a trusted subsystem, called, TRUCOUNT, that enables CFT-to-BFT transformations. Armed with the insights and TRUCOUNT, we develop a general methodology, called BUMBLEBEE, for transforming CFT protocols into BFT protocols. We apply BUMBLEBEE for transforming widely used CFT protocols, Raft, Paxos, and WPaxos into their hybrid counterparts. We demonstrate the feasibility of the BUMBLEBEE approach by transforming common CFT protocols including Raft, Paxos, and WPaxos into their hybrid counterparts. By incorporating our approach in etcd's Raft consensus, we show that overheads due to hybridization are less than 30% in terms of throughput. This dissertation shows that many existing CFT protocols can be easily enhanced to provide greater fault tolerance by incorporating a trusted component and changing some parts of the protocol. In doing so, we show that such transformations are simple enough that they can be easily understood, which may enable their greater practical adoption.

1.3 Summary of Proposed Post-Preliminary Exam Work

Posterior to the preliminary examination, this dissertation proposes to investigate methods to automate the BUMBLEBEE transformations. Given a specification of a CFT protocol in a formal language (e.g. TLA+ [55], Verdi [88]), an algorithm should recognize the key pieces of the protocol and produce a specification of the hybrid fault-tolerant version. Given the complexities implicit in many consensus protocols, such an approach would minimize the time required for such transformations and minimize unnecessary bugs that could be induced with manual methods. Furthermore, this dissertation proposes to investigate methods to auto-generate machine-checked proofs of correctness properties as well as high-performance implementations from the transformed specifications.

This dissertation also proposes to develop a highly scalable, completely decentralized, asynchronous byzantine fault tolerant protocol for cryptographic currency (cryptocurrency) and related applications. For such applications, prevention of double-spending, where an adversary transfers the same currency to multiple different users, is the core problem. Moreover, since consensus is decentralized, any number of participants are allowed, increasing the possibilities for Sybil attacks [41] – an adversary can create any number of pseudonyms and participate in processing transactions. Existing solutions overcome these problems by relying on proof-of-work based methods [86], but they exhibit significantly poor performance. Our intent with the proposed solution is to achieve consensus in seconds unlike other proof-of-work based solutions that solve consensus in the order of several minutes or even hours.

1.4 Dissertation Organization

This dissertation proposal is organized as follows. Chapter 2 provides the background for the contributions of this dissertation. Chapter 3 discusses the related previous works in the space. Chapter 4 describes and evaluates ezBFT. Chapter 5 describes and evaluates Dester. Chapter 6 describes and evaluates Bumblebee. Chapter 7 concludes this document and discusses proposed post-preliminary examination work.

Chapter 2

Background

This chapter provides the adequate background information for understanding the rest of this dissertation. Section 2.1 describes the problem of replication in distributed systems, while Section 2.2 explains the problem of consensus. Section 2.3 describes some of the prolific and relevant fault models employed in distributed systems. Finally, Section 2.4 compares and contrasts, with examples, three fault models, namely CFT, BFT, and Hybrids, that are important for understanding the contributions of this dissertation.

2.1 Replication

Replication is a fundamental technique that is widely adopted to implement fault-tolerance and improve the performance and scalability of today's online services. Performance is usually measured in terms of the throughput and the client perceived latency. As more and more services strive to reach customers globally, reducing the client perceived latency is of prime concern. To accomplish this, replicas are placed close to the clients. However, geographical scale replication is a challenge due to the high latency perceived between replicas. Many proposals in literature, such as [65, 72] and the contribution of this thesis, address this challenge. All the proposed replication techniques can be grouped into two broad categories [87]: Primary Copy and Update Everywhere.

In Primary Copy (aka. Passive) Replication, a master replica primarily serves client requests, while a set of slave replicas that synchronously/asynchronously receive the master replica's state and update their state. Most enterprise-grade databases today provide this feature out of the box [1, 10]. The disadvantages of this technique are twofold: (a) the master replica crash leads to the system's temporary unavailability until one of the slave replicas is elected as the master; (b) when the replication is asynchronous, the slave replicas do not necessarily reflect the master's most recent state, and thus when the master fails, the newly elected master will serve requests from an older state and requires manual intervention to correct the system state.

On the other hand, in Update Everywhere replication, there is no distinction between a primary and a copy replica. Thus, every node in the distributed system is responsible for executing client requests. One of the most widely adopted and studied Update Everywhere replication technique is State Machine Replication (aka. Active Replication) [80].

Under the state-machine approach, a distributed system is modeled as a set of states, and transition among states happen deterministically by executing client requests. Every replica in a distributed system must implement a state machine and have to be initialized with the same initial state. Thus, when the same sequence of client requests are executed on every replica, each one of them will reach the same final state. This technique is very effective in masking failures in a distributed system as every command issued to the system is executed by each replica. Therefore, if a replica fails (by crashing), the system does not fail because each replica maintains its own copy of the shared state and thus can make progress.

Maintaining a consistent copy of the state across all replicas entails the implementation of Consensus. The rest of the chapter focuses on Consensus and related algorithms, since the contribution of this dissertation addresses the problem of Consensus.

2.2 Consensus

Every replica of a distributed system is subject to numerous concurrent client requests. In order to keep the system consistent, replicas have to coordinate and agree on client requests that each node should execute, even in the presence of faults. This is widely known as the consensus problem. A typical consensus problem is defined by the following safety requirements [56]:

- *Nontriviality*: A request can only be agreed upon if it was proposed by a client.
- *Stability*: Once a request has been agreed for execution, no replica can revert its decision.
- *Consistency*: Two different replicas cannot agree on different request for execution.

2.3 Fault Models

A process executes the distributed algorithm assigned to it through the set of components implementing the algorithm within that process. A failure occurs whenever the process does not behave according to the algorithm. Our unit of failure is the process. When the process fails, all its components fail at the same time. Process abstractions differ according to the nature of the faults that cause them to fail. Possible failures range from a crash, where a process simply stops to execute any steps, over an omission to take some steps, a crash with subsequent recovery, to arbitrary and even adversarial behavior. We discuss these kinds of failures in the subsequent sections.

2.3.1 Crash Faults

Crash Fault is the simplest of the fault models, in which a process stop executing protocols steps. Under this model, a correct process is expected to execute the algorithm correctly and exchange messages with other processes in a timely manner for some time t. Beyond, the process may stop sending messages and executing the protocol to other processes. A process is said to be *faulty*, if it crashes some time during its execution; otherwise, it is said to be *correct*. Such protocols are said to be adopt the *crash-stop* model.

A consensus protocol under this fault model guarantees the safety properties by enforcing a limit on the number of processes that can be faulty at any given time. N denotes the total number of processes while f denotes the maximum number of tolerated faults. To guarantee safety, crash-fault tolerant protocols typically require $N = 2f + 1$ processes to tolerate f failures. Furthermore, to guarantee progress even under failures, such protocols depend on a quorum, instead of all, of the processes.

In practice, it is unlikely that no more than f processes during the entire execution of the protocol. Either crashed processes must be recovered or new processes must be be added to the protocol to ensure progress. Therefore, in practice, *crash-stop* protocols are enhanced with recovery mechanisms that help a crashed process rejoin the protocol after a restart. They are, hence, said to adopt the *crash-recovery* model. Typically, protocols in this fault model persist some state information to local stable storage to aid during the recovery procedure.

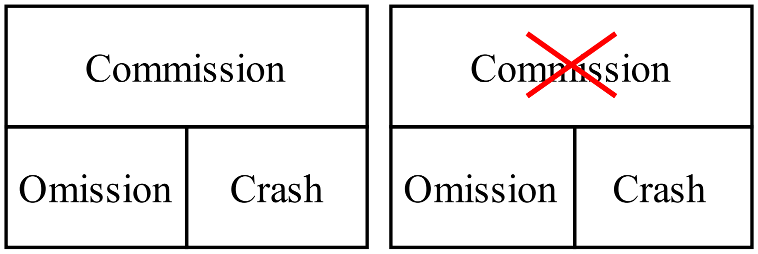
Some of the notable and prominent examples of protocols that tolerate crash faults are: Paxos [60], Viewstamped Replication [76], and Raft [77].

2.3.2 Byzantine Faults

Byzantine Faults is perhaps the most general of the fault models. Under this model, a process can behave *arbitrarily*; that is, it may deviate from the protocol specification. Protocols adopting this fault model are the most complex due to the many possible failure executions they must cope with.

Figure 2.1a illustrates the different kinds of failures in the byzantine failure model. Broadly, byzantine failures can be classified into *Omission* faults and *Commission* faults [33]. Under omission faults, a process may fail to send one or more messages specified by the protocol, but sends no incorrect messages. Crash faults are a subset of omission faults. Commission faults include all other non-omission faults in which a process may send a message not specified by the protocol. This includes *equivocation*, i.e., the ability of a process to make conflicting statements without being detected as having an intent to compromise consistency.

Because processes can equivocate, byzantine fault tolerant protocols require more than $N = 2f + 1$ processes to tolerate f failures. Specifically, N should be equal to $3f + 1$. The additional f replicas are required to overshadow the malicious acts (i.e., equivocation) of the byzantine replicas. Unlike CFT, protocols in the BFT model do not guarantee safety beyond f failures, thus it is important to proactively recover processes to avert any calamities [31].



(a) Byzantine Fault Model (b) Hybrid Fault Model

Figure 2.1: Failure hierarchy.

PBFT [30] is the first practical consensus protocol to tolerate Byzantine failures.

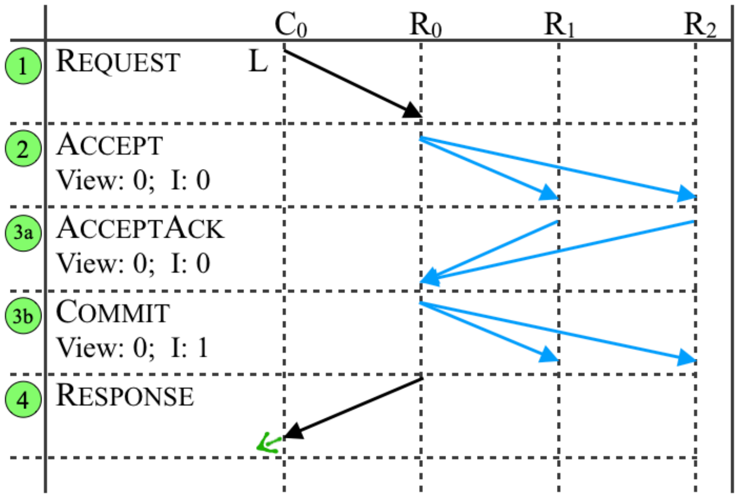
2.3.3 Hybrid Faults

The hybrid fault model is very similar to the byzantine fault model, except that the processes depend on a trusted subsystem to tolerate arbitrary faults. In this model (see Figure 2.1b), the ability of the replicas to equivocate is stripped away using a trusted subsystem [32, 61]. A trusted subsystem certifies the message sent by the replicas to ensure that a malicious replica cannot cause different correct replicas to execute different operations as their *i-th* operation. For example, in *Hybster* [26], replicas attest messages with *monotonically increasing counter certificates* using the `TrInX` trusted subsystem. Therefore, replicas cannot sign different messages using the same counter value without being detected.

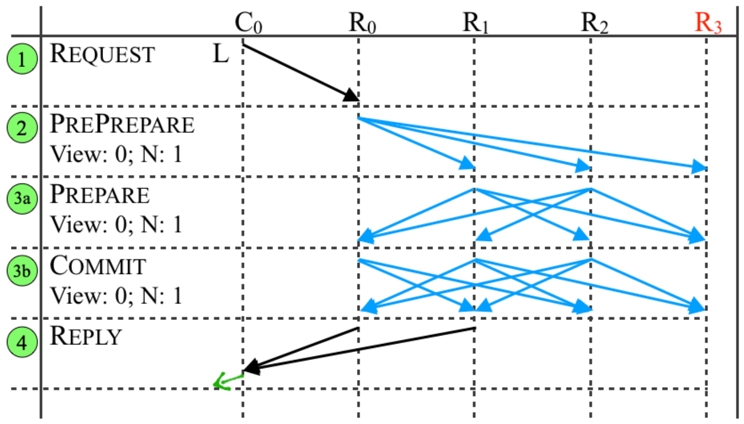
Thus, additional resources that were required to detect equivocation in the BFT model are unnecessary in the hybrid model. One fundamental change is that the number of replicas required to tolerate f failures reduces from $3f+1$ to $2f+1$. In the hybrid model, at most f replicas may behave arbitrarily with the exception of the trusted subsystem, which can only fail by crashing.

2.4 BFT vs CFT vs Hybrids

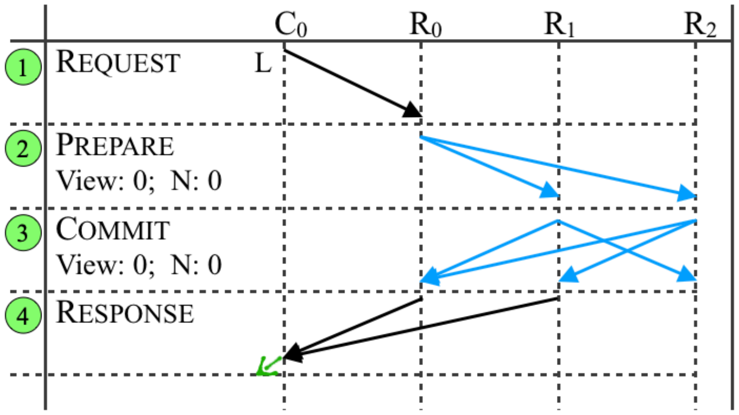
In this section, we highlight the fundamental design differences between the BFT, CFT, and the hybrid classes of protocols. We begin by comparing three example protocols, Paxos [60], PBFT [30], and Hybster [26], to illustrate their design differences and then proceed with a more general description.



(a) Paxos



(b) PBFT



(c) Hybster

Figure 2.2: An example of normal execution in three different failure models.

Paxos vs PBFT vs Hybster

We provide a walk-through of identical normal-case executions for each of the three protocols and highlight where each protocol performs uniquely. Note that we describe the three protocols hand-in-hand and only highlight the differences explicitly. A normal-case execution is finding an order for a client request, executing it in that order, and replying back to the client. Figure 2.2 provides the visual illustration. Paxos, PBFT, and Hybster are leader-based protocols, i.e., they designate a replica to propose the order for client commands.

(1) For each protocol, the execution starts when the client sends a request to the leader replica containing a command to be executed on the replicated state. For PBFT and Hybster, the client signs its requests to ensure that a potential byzantine leader does not tamper with the requests without detection.

(2) As soon as the leader receives the request, it assigns the command with a sequence number (*instance number* in Paxos and *order number* in Hybster), which defines the execution order. The leader `propose`s the command with the sequence number to all the replicas. PBFT certifies the message with a *message authentication code* (MAC), while Hybster certifies with a trusted subsystem-produced MAC. Also, note that in our example, Paxos and Hybster require three nodes each, while PBFT requires four.

The replicas receive the proposal from the leader.

(3) **Hybster** replicas acknowledge the proposal to each other in addition to the leader. Replicas wait for a majority of responses to commit and execute the command.

(3a) **Paxos** replicas reply back to the leader acknowledging the proposal. **PBFT** replicas exchange the proposal with each other to ensure that they received the same proposal from the leader (i.e., to ensure no equivocation). The proposal is validated if a majority of nodes respond with the same proposal from the leader.

(3b) **Paxos** leader, upon receipt of a majority of acknowledgments, commits the request, and sends a `commit` message to the replicas to do the same. **PBFT** replicas exchange commit messages. They execute the command upon collecting a majority quorum of these messages and reply to the client.

(4) **Paxos** client waits until it receives a reply from the leader. **PBFT** and **Hybster** clients wait until they receive identical replies from at least $f + 1$ replicas. This is because, waiting for only one potentially byzantine replica may provide an incorrect response.

The rest of this section generalizes the differences more broadly and presents some key insights.

Replicas

Consensus algorithms consist of two main components: agreement and leader election. We analyze the classes of protocols with respect to these two components. We note that consensus protocols designed for different fault models use different terminologies to define equivalent constructs; Table 2.1 summarizes the labels used in rest of this paper.

Leader Election or View Changes

One of the fundamental design differences between CFT and BFT protocols lie in how they elect a designated replica to propose commands, called *leader election* and *view change*, in the CFT and BFT models, respectively. Here, we only refer to single leader (or primary)-based protocols. During leader election in the CFT model, multiple replicas compete to gain leadership for a particular (logical) period. If an elected leader fails to make progress, another election takes place.

If such a scheme was applied in the BFT model, a malicious replica may repeatedly win an election but will fail to make progress (i.e., propose commands to the state machine). Thus, protocols in the BFT model use a different approach, also known as *view change.* Each logical period, during which a stable leader is active, is identified using a monotonically increasing number, called *view.* For each view number, the replica that will be responsible for proposing commands in that view is predefined. This ensures that a correct leader is established after at most f view changes, where f is the maximum number of byzantine replicas. In practice, the leader replica is obtained from the view number v using the formula $v \bmod N$, where N is the number of replicas.

Table 2.1: Summary of labels used by different protocols and their meaning.

Label	Meaning
View, Term, Ballot	Logical period in which the protocol makes progress.
Instance, Log Index, Order, Sequence	Positions in the replicated log to which client commands are assigned.
Leader, Primary, Owner	Distinguished node responsible for proposing order to client commands. An owner is the leader for a subset of client commands indicated by objects they access.
View Change, Leader Election, Term Change	Process of replacing a leader/primary with another one.

Agreement

Once a stable leader is established, the leader starts proposing client commands to be applied on the replicated state. In a CFT protocol, this agreement process usually consists of two phases: (i) the leader proposes a command to be replicated to other replicas, and replicas acknowledge the receipt of the command; and (ii) the leader finalizes the command to all replicas after acknowledgment. The last phase is unnecessary if the replicas simply acknowledge each other in addition to the leader. This is an optimization that many CFT protocols implement to minimize communication steps (and reduce client-perceived latency) [60, 73, 78]. For example, in Figure 2.2a, the `Commit` step can be removed; instead, replicas broadcast their `AcceptAck` messages to each other and a quorum of those messages constitute a commit indication.

However, for BFT protocols, replicas acknowledging to each other is not an optimization but a requirement. Since the replicas including the leader can be byzantine, messaging with each other is the only way for correct replicas to detect equivocation. Thus, generally BFT protocols perform two such acknowledgements: one to ensure that the replicas have received the same proposal from the leader, and a second to acknowledge that a majority of the replicas agree with that proposal.

Hybrid protocols also need to perform all-to-all message exchanges, but only once. Due to the equivocation prevention mechanism, replicas can be certain that the leader cannot equivocate without detection. Thus, the most a leader can equivocate is to propose the same command twice with two different attestations, but a correct replica can detect this using the metadata of the last signed client request.

Client

In the CFT model, the client sends a request to one of the replicas and waits until it receives a reply with the result of the operation. In contrast, in the BFT model, the client sends a request to one of the replicas, but expects a reply from at least $f + 1$ of the replicas. This is because, the client cannot trust a single reply since it may be from a byzantine replica. Therefore, a client must wait until it receives $f + 1$ matching replies to validate the result. Due to the assumption of byzantine replicas in the hybrid fault model, the clients in this model use the same reply validation mechanism.

A property that is unique to the BFT and hybrid models is that the client messages are signed in the BFT model using the client's private key in order to prevent byzantine replicas from tampering with commands that should be executed by the state machine.

Chapter 3

Related Work and System Model

3.1 Related Work

This section presents state-of-the-art in the literature relevant to the contributions of this thesis. For clarity, we group the related works by their fault models. Furthermore, we also describe the state-of-the-art leaderless protocols that inspire ezBFT (Chatper 4) and Dester (Chapter 5).

3.1.1 Byzantine Fault Tolerant Protocols

BFT consensus was first introduced in [59]. However, PBFT [30, 31] was the first protocol to provide a practical implementation of BFT consensus in the context of state machine replication. PBFT solved consensus in three communication steps (excluding two steps for client communication) using $3f+1$ nodes and requiring responses from at least $2f+1$ nodes. f is the maximum number of byzantine faulty nodes that the system can tolerate and make progress.

FaB [66] reduced the number of communication steps required to reach agreement in the common case to two steps, called two-step consensus (excluding two steps for client communication). However, node and quorum requirements are substantially larger with $5f+1$ and $4f+1$ nodes, respectively, to tolerate f faults and still achieve two-step consensus. FaB falls back to a slower path when $4f+1$ responses cannot be acquired, and requires an extra communication step and at least $3f+1$ nodes to reach agreement.

FaB was the first BFT protocol to not require any digital signatures in the common case. Parameterized-FaB [66] requires $3f+2t+1$ nodes, where f is the number of tolerated faults and preserve safety and t is the number of tolerated faults and solve two-step consensus. This minimized the node requirements in the common case to $3f+1$ (by setting $t=0$), but the quorum requirement is still more than that of PBFT.

The Q/U [12] protocol was the first to achieve consensus in two communications steps when there are no faults and update requests do not concurrently access the same object. Q/U defines a simplified version of conflicts. Requests are classified as either reads or writes. Reads do not conflict with reads, while write conflicts with reads and writes. This is more restrictive than the commutative property used by ezBFT. In ezBFT, for instance, mutative operations (such as incrementing a variable) are commutative.

HQ [38] is similar to PBFT with a special optimization to execute read-only requests in two communication steps and update requests in four communication steps under no conflicts. HQ's definition of conflict is the same as Q/U's.

Zyzzyva [52, 53] uses $3f+1$ nodes to solve consensus in three steps (including client communication), requiring a quorum of $3f+1$ responses. The protocol tolerates f faults, so it takes an additional two steps when nodes are faulty. Zyzzyva uses the minimum number of nodes, communication steps, and one-to-one messages to achieve fast consensus. It is cheaper than the aforementioned protocols, but also more complex. Zyzzyva's performance boost is due to speculative execution, active participation of the clients in the agreement process, and tolerating temporary inconsistencies among replicas.

ezBFT has the same node and quorum requirements as well as the number of communication steps as Zyzzyva. However, by minimizing the latency of the first communication step and alleviating the specialized role of the primary, ezBFT reduces the request processing latency.

Aliph [45] builds a BFT protocol by composing three different sub-protocols, each handling a specific system factor such as contention, slow links, and byzantine faults. Under zero contention, the sub-protocol *Quorum* can deliver agreement in two steps with $3f + 1$ nodes by allowing clients to send the requests directly to the nodes. However, as contention or link latency increases, or as faults occur, Aliph switches to the sub-protocol *Chain* whose additional steps is equal to the number of nodes in the system, or to the sub-protocol *Backup* which takes at least three steps. Although the idea of composing simpler protocols is appealing in terms of reduced design and implementation complexities, the performance penalty is simply too high, especially in geo-scale settings.

In contrast, ezBFT exploits the trade-off between the slow and fast path steps. ezBFT uses three steps compared to Aliph's two steps in the common case, and in return, provides slow path in only two extra communication steps unlike Aliph. Moreover, ezBFT's leaderless approach reduces the latency of the first communication step to near zero, yielding client-side latency comparable to Aliph's.

EBAWA [83] uses the spinning primary approach [84] to minimize the client-side latency in geo-scale deployments. However, a byzantine replica can delay its commands without detection reducing the overall server-side throughput. ezBFT's dependency collection mechanism enables correct replicas to only depend on commands that arrive in time, and execute without waiting otherwise.

Table 3.1 summarizes the comparison of existing work with ezBFT. Note that ezBFT and Zyzzyva have the same best-case communication steps. However, for ezBFT, the latency for the first-step communication is minuscule (tending towards zero) when compared to Zyzzyva's first-step latency.

Table 3.1: Comparison of existing BFT protocols and EZBFT.

Protocol	**PBFT**	**Zyzzyva**	**Aliph**	**ezBFT**
Resilience	$f < n/3$	$f < n/3$	$f < n/3$	$f < n/3$
Best-case comm. steps	5	3	2	3
Best-case comm. steps in absence of ...	Byz. Slow links	Byz. Slow links Contention	Byz. Slow links Contention	Byz. Slow links Contention
Slow-path steps	-	2	n + 3	2
Leader	Single	Single	Single	Leaderless

3.1.2 Hybrid Fault Tolerant Protocols

A multitude of works in the past have proposed the use of trusted hardware to provide increased fault tolerance at lower cost in distributed systems. A2M [32] is one of the first efforts to show that a small trusted component can significantly improve security in distributed systems. A2M uses an append-only log backed by a trusted module and demonstrated its use with A2M-PBFT, a PBFT variant with $2f+1$ node requirement. Due to the high overheads of maintaining the append-only log that can grow indefinitely, TrInc [61] was proposed as a smaller and simpler alternative to A2M. TrInc proposes a monotonically increasing trusted counter backed by a key. BUMBLEBEE uses a variant of the TrInc subsystem.

Efforts such as [51, 85] reduce the complexity of existing BFT protocols using trusted hardware. MinBFT and MinZyzzyva [85] are protocols obtained by incorporating a trusted counter implementation, called *USIG*, into PBFT and Zyzzyva [52], respectively. CheapBFT [51] uses a *CASH* subsystem that is implemented using FPGAs. In contrast, we argue that BUMBLEBEE is the first effort to use a trusted subsystem to improve the fault tolerance of CFT protocols.

Our Hybrid Paxos is functionally equivalent to Hybster [26], but significant differences underlie their designs. Hybster is a hybrid fault tolerant protocol that is designed from the ground up. In contrast, Hybrid Paxos is designed through a transformation of Paxos. We also present a general method to transform CFT protocols into BFT by leveraging a trusted counter implementation. In doing so, we do not focus on boosting the protocol performance as Hybster does with its *consensus-oriented parallelism* approach. Different systems use consensus differently – Spanner [7, 35] and CockroachDB [4] use a separate consensus protocol per shard of data, while FaunaDB uses a single, global consensus protocol [11]. Hence, we leave it to the system builders to implement specific optimizations to cater to their use cases.

Due to the high overheads of BFT protocols, past efforts have used special consistency verification processes [68] and/or fault-recovery mechanisms [14] to detect or tolerate non-crash, non-malicious failures in CFT systems. However, this unnecessarily adds complexity to layers outside of the core consensus layer. The Bumblebee approach is more appropriate as a design choice when such complexity is unwarranted; it allows system builders to transform well-understood CFT protocols, in a straightforward way, into their hybrid versions.

Table 3.2 summarizes the comparison of existing work with Dester. Note that Dester and Hybster have the same best-case communication steps. However, for Dester, the latency for a client to send a command to a replica is minuscule (tending towards zero) when compared to Hybster's latency. Furthermore, even though TruDep is more complex some subsystems such as TrInX, it does not require any checkpointing mechanism as A2M [32] requires. Moreover, the TEE-based design makes use of software data structures such as map to efficiently implement the component in fewer lines of code. Thus, unlike systems such as CASH [51], TruDep's codebase is still small (less than 250 lines of core logic).

Table 3.2: Comparison of existing BFT protocols and Dester.

Protocol	**PBFT**	**A2M**	**MinBFT, CheapBFT, TrInc**	**Hybster**	Dester
Resilience	$f < n/3$	$f \leq n/2$	$f \leq n/2$	$f \leq n/2$	$f \leq n/2$
Trusted Component	No	Yes (Virtual machine)	Yes (ASIC/FPGA)	Yes (CPU TEEs)	Yes (CPU TEEs)
Trusted Component Complexity	-	High	Low	Low	Medium
Best-case comm. steps	3	3	2	2	2
Best-case comm. steps in absence of...	Byz. Slow links	Byz. Slow links	Byz. Slow links	Byz. Slow links	Byz. Slow links Contention
Slow-path steps	-	-	-	-	1
Leader	Single	Single	Single	Single	Leaderless

3.1.3 Cross Fault Tolerant Protocols

In [64], the authors present *cross fault tolerance* (XFT), a fault tolerance model that only addresses certain classes of byzantine faults. XPaxos [64], the first protocol to implement the XFT model, can tolerate t byzantine failures in a $2t + 1$ system and provide a resilient service as long as a majority of replicas are correct and communicate synchronously. As the authors point out, XFT is suitable in geographically distributed systems, where an adversary cannot compromise t replicas and cause network partitions among correct ones at the same time. Our approach can tolerate network partitions and t byzantine replicas at the same time, as long as the trusted component is not compromised.

3.1.4 CFT-to-BFT Transformation

We are not the first to present a technique for transforming CFT protocols into BFT protocols. Nysiad [46] translates byzantine faults into crash faults by adding additional nodes, called *guards*, which detect and prevent byzantine behavior. Each node requires at least $3f + 1$ guards to tolerate f byzantine failures. In contrast, the BUMBLEBEE approach requires only $2f+1$ replicas. Furthermore, [46] uses a complex attestation mechanism to verify the integrity of sent messages. The use of the trusted component in BUMBLEBEE greatly simplifies message attestations and prevents equivocation.

In [58], Lamport presented a transformation of the Paxos protocol to tolerate byzantine faults by adding f additional replicas and modifying both phases of the protocol. The resulting protocol was very similar to PBFT [30, 31], with high overheads than CFT protocols. In contrast, our intent is to achieve greater fault tolerance in existing CFT protocols, without the need for additional communication steps and message exchanges, thereby reducing overheads and also achieving high performance.

3.1.5 Leaderless Consensus

Leaderless and multi-leader protocols [20, 65, 73, 78] have been proposed for the CFT model.

EPaxos [72] is a multi-leader consensus protocol that, unlike Mencius, exhibits high availability despite replica crashes as long as a majority of the nodes are up and running. The protocol works in two phases: a fast phase that is reached under no contention and an additional slow phase that is required under contention. Contention is defined as the percentage of requests that contend for (or access) the same set of objects at the same time. In EPaxos, the ordering is detached from the command execution. There is a separate graph-based dependency linearization mechanism that is adapted to define the final order of execution of commands. As soon as a command c is committed, every replica builds a dependency graph by adding c and its dependencies. The next step is finding the strongly connected components and sorting them in reverse topological order. Once the commands in each strongly connected components are sorted according to their sequence number seq, they are executed one by one.

Similarly, Caesar also works in two phases and uses a dependency collection mechanism similar to EPaxos, but avoids the expensive graph processing phase by incorporating the linearization phase within the protocol's critical path, all without incurring any additional overhead. Moreover, even under contention, Caesar's novel fast decision scheme is optimized to increase the probability to decide in two communication delays even in those scenarios. Caesar can deliver fast phase consensus even under non-trivial contention by having a replica wait until some conditions are satisfied before replying to the primary. However, such wait conditions are harmful in BFT protocols, because a malicious replica can use this as an opportunity to cease progress.

Alvin [81] is also a multi-leader consensus protocol similar to EPaxos, but, unlike EPaxos, it is able to avoid the expensive computation on the dependency graphs enforced by EPaxos via a decision slot-centric approach *à la* Mencius. Alvin decides the order of a request after two communication delays under no conflicts. However, it still suffers from the same vulnerability to conflicts of EPaxos: a command's leader is not able to decide the command on a fast path if it observes discordant opinions about the command from a quorum of nodes, and thus incurs two more communication delays (four, in total) to decide. Caesar overcomes this problem as mentioned previously.

In M^2Paxos [78], a replica can order a request if it *owns* the object that the request accesses. Otherwise, it forwards the request to the right owner or acquires ownership. Acquiring ownership means becoming the primary for some subset of objects, and in CFT-based protocols, any replica can propose to be a owner of any subset of objects at any point in time. However, in BFT-based protocols, electing a primary is a more involved process requiring consent from other replicas. In addition, view numbers are pre-assigned to replicas; therefore, randomly choosing primaries is not a trivial process.

3.2 System Model

This section presents the system model and assumptions behind the contributions of this dissertation. We consider set of nodes (replicas and clients), in an asynchronous system, that communicate via message passing. The replica nodes have identifiers in the set $\{R_0, ..., R_{N-1}\}$.

ezBFT

We assume the byzantine fault model in which nodes can behave arbitrarily. We also assume a strong adversary model in which faulty nodes can coordinate to take down the entire system. Every node, however, is capable of producing cryptographic signatures [50] that faulty nodes cannot break. A message m signed using R_i's private key is denoted as $\langle m \rangle_{R_i}$. The network is fully connected and quasi-reliable: if nodes R_1 and R_2 are correct, then p_2 receives a message from R_1 exactly as many times R_1 sends it.

To preserve safety and liveness, ezBFT requires at least $N = 3f + 1$ replica nodes in order to tolerate f Byzantine faults. ezBFT uses two kinds of quorums: a fast quorum with $3f+1$

replicas, and a slow quorum with $2f+1$ replicas. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication.

Dester

We assume the hybrid fault model in which nodes can behave arbitrarily, except the trusted subsystem that can only fail by crashing. Every node, however, is capable of producing cryptographic signatures [50] that faulty nodes cannot break. The network is fully connected and quasi-reliable: if nodes R_1 and R_2 are correct, then p_2 receives a message from R_1 exactly as many times R_1 sends it.

To preserve safety and liveness, Dester requires at least $N = 2f+1$ replica nodes in order to tolerate f arbitrary faults. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication.

Bumblebee

Now, we describe the system model and assumptions for the Bumblebee transformation approach. We consider a system of nodes (or replicas) that communicate through message passing as specified by a consensus protocol. We refer to the protocols before and after transformation as, *original* and *transformed*, respectively. The original protocol is only crash fault tolerant, while the transformed protocol is hybrid fault tolerant. We achieve hybrid fault tolerance with the help of trusted execution environments or enclaves (e.g. Intel SGX), as described in Section 2.

We consider an adversary that controls all the system software including the operating system. Thus, the adversary can schedule multiple instances of the same enclave, offer the latest and previous versions of sealed data, and block, delay, and read and modify all messages sent by the enclaves. However, the adversary cannot read or modify the enclave runtime memory or learn any information about the secrets held in enclave data. Furthermore, the enclave is capable of generating cryptographic operations that the adversary cannot break. We also assume that the adversary cannot compromise the SGX protections on participating nodes (e.g. via physical attacks). We acknowledge that the persistent state maintained by the enclaves are susceptible to rollback attacks; however, in Chapter 6.5, we show how our transformed protocols intrinsically prevent such attacks.

To guarantee safety, we assume that at most f replicas are faulty, at any given time, in a system of $N = 2f+1$ replicas.

Chapter 4

ezBFT

This chapter presents the design and implementation of ezBFT, a leaderless consensus protocol capable of tolerating Byzantine faults. To the best of our knowledge, ezBFT is the first BFT protocol to provide decentralized, deterministic consensus in the eventually synchronous model. ezBFT enables every replica in the system to process the requests received from the clients. Doing so (i) significantly reduces the client-side latency, (ii) distributes the load across replicas, and (iii) tolerates faults more effectively. Most importantly, ezBFT delivers requests in *three* communication steps in normal operating conditions. Furthermore, by minimizing the latency at each communication step, ezBFT provides a highly effective BFT solution for geo-scale deployments. ezBFT has been formally specified in TLA+ and model checked using the TLC model checker tool.

The rest of the chapter is organized as follows. Section 4.1 overviews ezBFT, and Section 4.3 presents a complete algorithmic design and correctness arguments. An experimental evaluation of ezBFT is presented in Section 4.4.

4.1 Overview

Geographical-scale (or "geo-scale") deployments of BFT systems have an additional challenge: achieving low client-side latencies and high server-side throughput under the high communication latencies of a WAN. Since replicas need to communicate with each other and the clients to reach consensus, the number of communication steps incurred directly impacts the latency, as each step involves sending messages to potentially distant nodes. Thus, protocols such as Zyzzyva [52], Q/U [66], and HQ [38] use various techniques to reduce communication steps. These techniques, however, do not reduce client-side latencies in a geo-scale setting, where, the latency per communication step is as important as the number of communication steps. In other words, a protocol can achieve significant cost savings if the latency incurred during a communication step can be reduced.

ezBFT can deliver decisions in three communication steps from the client's point-of-view, if there is no contention, no byzantine failures, and synchronous communication between replicas. The three communication steps include: (i) a client sending a request to any one of the replicas (closest preferably); (ii) a replica forwarding the request to other replicas with a proposed order; and (iii) other replicas (speculatively) executing the request as per the proposed order and replying back to the client. These three steps constitute ezBFT's core novelty. To realize these steps, ezBFT incorporates a set of techniques summarized below and explain in detail in Section 4.3.

First, the ezBFT replicas perform speculative execution of the commands upon receiving the proposal messages from their respective command-leaders (see below). With only one replica-to-replica communication, there is no way to guarantee the final commit order for client commands. Thus, the replica receiving the proposal assumes that other replicas received the same proposal (i.e., the command-leader is not byzantine) and that they have agreed to the proposal. With this assumption, replicas speculatively execute the commands on their local state and return a reply back to the client.

Second, in ezBFT, the client is actively involved in the consensus process. It is responsible for collecting messages from the replicas and ensuring that they have committed to a single order before delivering the reply. The client also enforces a final order, if the replicas are found to deviate.

Third, and most importantly, there are *no* designated primary replicas. Every replica can receive client requests and propose an order for them. To clearly distinguish the replica proposing an order for a command from other replicas, we use the term *command-leader*. A *command-leader* is a replica that proposes an order for the commands received from its clients. For clarity, all replica can be command-leaders. To ensure that client commands are consistently executed across all correct replicas, ezBFT exploits the following concepts.

ezBFT uses the concept of *command interference* to empower replicas to make independent commit decisions. If replicas concurrently propose commands that do not interfere, they can be committed and executed independently, in parallel, in any order, and without the knowledge of other non-interfering commands. However, when concurrent commands do interfere, replicas must settle on a common sequential execution.

4.2 Preliminaries

- **Command Interference**: A command encapsulates an operation that must be executed on the shared state. We say that two commands L_0 and L_1 are interfering if the execution of these commands in different orders on a given state will result in two final states. That is, if there exists a sequence of commands Σ such that the serial execution of Σ, L_0, L_1 is not equivalent to Σ, L_1, L_0, then L_0 and L_1 are interfering.

- **Instance Space**: An instance space can be visualized as a sequence of numbered slots to which client-commands can be associated with. The sequence defines the execution order of requests, and the role of a consensus protocol is to reach agreement among a set of replicas on a common order. However, to accommodate our requirement that every replica can be a command-leader for their received requests, each replica has its own instance space.

Thus, ezBFT's role is not only to reach consensus on the mapping of client-commands to the slots within an instance space, but also among the slots in different instance spaces.

- **Instance Number** An instance number, denoted I, is a tuple of the instance space (or replica) identifier and a slot identifier.

- **Instance Owners** An owner number, O, is a monotonically increasing number that is used to identify the owner of an instance space. Thus, there are as many owner numbers as there are instance spaces (or replicas). This number becomes useful when a replica is faulty, and its commands must be recovered by other replicas. When replicas fail, another correct replica steps up to take ownership of the faulty replica's instance space. The owner of a replica R_0's instance space can be identified from its owner number using the formula O_{R0} mod N, where N is the number of replicas. Initially, the owner number of each instance space is set to the owner replica's identifier (e.g., $O_{R0} = 0$, $O_{R1} = 1$, and so on).

- **Dependencies** Due to the use of per-replica instance spaces, the protocol must agree on the relationship between the slots of different instances spaces. ezBFT does this via dependency collection, which uses the command interference relation. The dependency set $\mathcal{D}$ for command L is every other command L' that interferes with L.

- **Sequence Number ($\mathcal{S}$)** is a globally shared number that is used to break cycles in dependencies. It starts from one and is always set to be larger than all the sequence numbers of the interfering commands. Due to concurrency, it is possible that interfering commands originating from different command-leaders are assigned the same sequence number. In such cases, the replica identifiers are used to break ties.

Protocol Properties

ezBFT has the following properties:

1. **Nontriviality.** Any request committed and executed by a replica must have been proposed by a client.
2. **Stability.** For any replica, the set of committed requests at any time is a subset of the committed requests at any later time. If at time t_1, a replica R_i has a request L committed at some instance I_L, then R_i will have L committed in I_L at any later time $t_2 > t_1$.
3. **Consistency.** Two replicas can never have different requests committed for the same instance.
4. **Liveness.** Requests will eventually be committed by every non-faulty replica, as long as at least $2f + 1$ replicas are correct.

4.3 Design

In this section, we present EZBFT in detail, along with an informal proof of its properties. We have also developed a TLA+ specification of EZBFT and model-checked the protocol correctness; this can be found in the technical report [21].

A client command may either take a *fast path* or a *slow path*. The fast path consists of three communication steps, and is taken under no contention, no byzantine failures, and during synchronous communication periods. The *slow path* is taken otherwise to guarantee the final commit order for commands, and incurs two additional steps.

4.3.1 The Fast Path Protocol

The fast path consists of three communication steps in the critical path and one communication step in the non-critical path of the protocol. Only the communication steps in the critical path contribute to the client-side latency. The fast path works as follows.

1. Client sends a request to a replica.

The client c requests a command L to be executed on the replicated state by sending a message $\langle \text{REQUEST}, L, t, c\rangle_{\sigma_c}$ to a EZBFT replica. The closest replica may be chosen to achieve the optimal latency. The client includes a timestamp t to ensure exactly-once execution.

2. Replica receives a request, assigns an instance number, collects dependencies and assigns a sequence number, and forwards the request to other replicas.

When a replica R_i receives the message $m = \langle \text{REQUEST}, L, t, c\rangle_{\sigma_c}$, it becomes the command-leader for L. It assigns c to the lowest available instance number I_L in its instance space and collects a dependency set $\mathcal{D}$ using the command interference relation that was previously described. A sequence number $\mathcal{S}$ assigned for c is calculated as the maximum of sequence numbers of all commands in the dependency set. This information is relayed to all other replicas in a message $\langle\langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d\rangle_{\sigma_{Ri}}, m\rangle$, where $d = H(m)$, h is the digest of R_i's instance space, and O_{Ri} is its owner number.

Nitpick. Before taking the above actions, R_i ensures that the timestamp $t > t_c$, where t_c is the highest time-stamped request seen by R_i thus far. If not, the message is dropped.

3. Other replicas receive the SPECORDER message, speculatively executes the command according to its local snapshot of dependencies and sequence number, and replies back to the client with the result and an updated set of dependencies and sequence number, as necessary.

When replica R_j receives a message $\langle\langle \textsc{SpecOrder}, O_{Ri},\ I_L, \mathcal{D}, \mathcal{S}, h, d\rangle_{\sigma_{Ri}}, m\rangle$ from replica R_i, it ensures that m is a valid REQUEST message and that $I_L = maxI_{Ri} + 1$, where $maxI_{Ri}$ is the largest occupied slot number in R_i's instance space. Upon successful validation, R_j updates the dependencies and sequence number according to its log, speculatively executes the command, and replies back to the client. A reply to the client consists of a message $\langle\langle \textsc{SpecReply},\ O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t\rangle_{\sigma_{Ri}}, R_j, rep, SO\rangle$, where rep is the result, and $SO = \langle \textsc{SpecOrder}, O_{Ri}, I_L,\ \mathcal{D}, \mathcal{S}, h, d\rangle_{\sigma_{Ri}}$.

4. The client receives the speculative replies and dependency metadata.

The client receives messages $\langle\langle \textsc{SpecReply}, O, I_L, \mathcal{D}', \mathcal{S}', d, c, t\rangle_{\sigma_{Rk}}, R_k, rep, SO\rangle$, where R_k is the sending replica. The messages from different replicas are said to match if they have identical O, I_L, $\mathcal{D}'$, $\mathcal{S}'$, c, t, and rep fields. The number of matched responses decides the fast path or slow path decision for command L.

4.1 The client receives $3f + 1$ matching responses.

The receipt of $3f + 1$ matching responses from the replicas constitutes a fast path decision for command L. This happens in the absence of faults, network partitions, and contention. The client returns reply rep to the application and then asynchronously sends a message $\langle \textsc{CommitFast}, c, I_L, CC\rangle$, where CC is the commit certificate consisting of $3f+1$ matching SPECREPLY responses, and returns.

5. The replicas receive either a COMMITFAST or a COMMIT message.

5.1 The replicas receive a COMMITFAST message.

Upon receipt of a $\langle \textsc{CommitFast}, c, I_L, CC\rangle$ message, the replica R_i marks the state of L as committed in its local log and enqueues the command for final execution. The replica does not reply back to the client.

Example. Figure 4.1 shows an example case. The client sends a signed REQUEST message to replica R_0 to execute a command L_0 on the replicated service. Replica R_0 assigns the lowest available instance number in its instance space to L_0. Assuming that no instance number was used previously, the instance number assigned to L_0 is $I_{L_0} = \langle r_0, 0\rangle$. Then, R_0 collects dependencies and assigns a sequence number to L_0. As the first command, there exists no dependencies, so the dependency set $\mathcal{D} = \{\}$. Thus, the sequence number is $\mathcal{S} = 1$.

A signed SPECORDER message is sent to other replicas in the system with the command and compiled metadata. Other replicas – R_1 through R_3 – receive this message, add the command to their log, and start amassing dependencies from their log that are not present in $\mathcal{D}$. No other replica received any other command, thus they produce an empty dependency

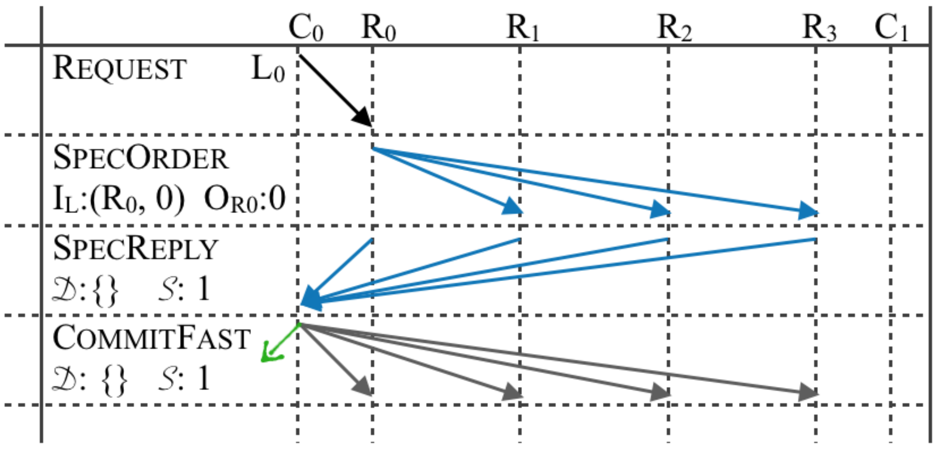


Figure 4.1: An example of a fast path execution.

set as well, and the sequence number remains the same. Since there are no dependencies, all replicas immediately execute the command, speculatively, on their copy of the application state. The result of execution, unchanged dependency set, sequence number, and the digest of log are sent in a SPECREPLY message to the client. The client checks for identical replies and returns the result to the application. The replies are identical in this case because no other command conflicts with L_0 in any of the replicas and the replicas are benign.

4.3.2 Execution Protocol

EZBFT uses *speculative execution* as a means to reply to the client quickly. However, the protocol must ensure that every correct replica has identical copies of the state. This means that, when necessary (as described in Section 4.3.3), the speculative state must be rolled back and the command must be re-executed correctly; this is called *final execution.*

Moreover, differently from existing BFT solutions, EZBFT collects command dependencies that form a directed graph with potential cycles. The graph must be processed to remove cycles and obtain the execution order for a command.

Each replica takes the following steps:

1. Waits for the command to be enqueued for execution. For final execution, wait for the dependencies to be committed and enqueued for final execution as well.
2. A dependency graph is constructed by including R and all its dependencies in $\mathcal{D}$ as nodes and adding edges between nodes indicating the dependencies. The procedure is repeated recursively for each dependency.
3. Strongly connected components are identified and sorted topologically.
4. Starting from the inverse topological order, every strongly connected component is identified, and all the requests within the component are sorted in the sequence num-

ber order. The requests are then executed in the sequence number order, breaking ties using replica identifiers. During speculative execution, the execution is marked *speculative* on the shared state. During final execution, the speculative results are invalidated, command re-executed, and marked *final*.

Note that speculative execution can happen in either the *speculative* state or in the *final* version of the state, which ever is the latest. However, for final execution, commands are executed only on the previous *final* version.

4.3.3 The Slow Path Protocol

The slow path is triggered whenever a client receives either unequal and/or insufficient SpecReply messages that is necessary to guarantee a fast path. The client will receive unequal replies if the replicas have different perspectives of the command dependencies, possibly due to contention or due to the presence of byzantine replicas. The case of insufficient replies happen due to network partitions or byzantine replicas.

The steps to commit a command in the slow path are as follows.

4.2 The client receives at least $2f + 1$ possibly unequal responses.

The client c sets a timer as soon as a Request is issued. When the timer expires, if c has received at least $2f + 1$ $\langle\langle \text{SpecReply}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, d, c, t\rangle_{\sigma_{Ri}}, R_j, rep, SO\rangle$ messages, it produces the final dependency set and sequence number for L. The dependency sets from a known set of $2f+1$ replicas are combined to form $\mathcal{D}'$. A new sequence number $\mathcal{S}'$ is generated if the individual dependency sets were not equal. c sends a $\langle \text{Commit}, c, I_L, \mathcal{D}', \mathcal{S}', CC\rangle_{\sigma_c}$ message to all the replicas, where CC is the commit certificate containing $2f+1$ SpecReply messages that are used to produce the final dependency set.

Nitpick. Each command-leader specifies a known set of $2f+1$ replicas that will form the slow path quorum, which is used by the client to combine dependencies when more than $2f + 1$ reply messages are received. This information is relayed to the clients by the respective command-leaders and is cached at the clients.

5.2 The replicas receive a Commit message.

Upon receipt of a $\langle \text{Commit}, c, I_L, \mathcal{D}', \mathcal{S}', CC\rangle_{\sigma_c}$ message, replica r updates command L's metadata with the received dependency set $\mathcal{D}'$ and sequence number $\mathcal{S}'$. The state produced after the speculative execution of L is invalidated, and L is enqueued for final execution. The result of final execution, rep is sent back to the client in a $\langle \text{CommitReply}, L, rep\rangle$ message.

6.2 The client receives $2f + 1$ CommitReply messages and returns to the application.

The client returns *rep* to the application upon receipt of $2f+1$ $\langle \textsc{CommitReply}, L, rep\rangle$ messages. At this point, execution of command L is guaranteed to be safe in the system, while tolerating upto f byzantine failures. Moreover, even after recovering from failures, all correct replicas will always execute L at this same point in their history to produce the same result.

Example

Figure 4.2 shows an example of a slow path. Two clients c_0 and c_1 send signed REQUEST messages to replicas R_0 and R_3, respectively, to execute commands L_1 and L_2, respectively, on the replicated service. Assume that L_1 and L_2 conflict. Replica R_0 assigns the lowest available instance number of $\langle R_0, 0\rangle$ to L_1. Thus, R_0 collects a dependency set $\mathcal{D}_{L_1} = \{\}$ and assigns a sequence number $\mathcal{S}_{L_1} = 1$ to L_1. Meanwhile, R_3 assigns the instance number $\langle R_3, 0\rangle$ to L_1; the dependency set is $\mathcal{D}_{L_2} = \{\}$, and sequence number is $\mathcal{S}_{L_2} = 1$. Replicas R_0 and R_3 send SPECORDER messages with their respective commands and their metadata to other replicas. Let's assume that R_0 and R_1 receive L_1 before L_2, while R_2 and R_3 receive L_2 before L_1. The dependency set and sequence number will remain unchanged for L_1 at R_0 and R_1, because the dependency set in the SPECORDER message received is the latest. However, the dependency set and sequence number for L_2 will update to $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$, respectively. Similarly, the dependency set and sequence number will remain unchanged for L_2 at R_0 and R_1, but for L_1, $\mathcal{D}'_{L_1} = \{L_2\}$ and $\mathcal{S}'_{L_1} = 2$, respectively. The SPECREPLY messages for both L_1 and L_2 with the new metadata are sent to the respective clients c_0 and c_1 by the replicas.

Let's assume that the slow quorum replicas are R_0, R_1, and R_2 for R_0, and R_1, R_2, and R_3 for R_3. Since client c_0 observes unequal responses, it combines the dependencies for L_1 from the slow quorum and selects the highest sequence number to produce the final dependency set $\mathcal{D}_{L_1} = \{L_2\}$ and sequence number $\mathcal{S}_{L_1} = 2$. This metadata is sent to the replicas in a COMMIT message. Similarly, c_1 produces the final dependency set $\mathcal{D}_{L_2} = \{L_1\}$ and sequence number $\mathcal{S}_{L_2} = 2$ for L_2, and sends a COMMIT message to the replicas. The replicas update the dependency set and sequence number of the commands upon receipt of the respective COMMIT messages, and the commands are queued for execution. The replicas wait for the receipt of the COMMIT messages of all commands in the dependency set before processing them.

After the construction of the graph and the inverse topological sorting, there will exist commands L_1 and L_2 with a cyclic dependency between them. Since the sequence numbers for both the commands are the same and thus cannot break the dependency, the replica IDs are used in this case. Thus, L_1 gets precedence over L_2. L_1 is executed first, followed by L_2. The result of the executions are sent back to the clients. The clients collect $2f+1$ reply messages and return the result to the application.

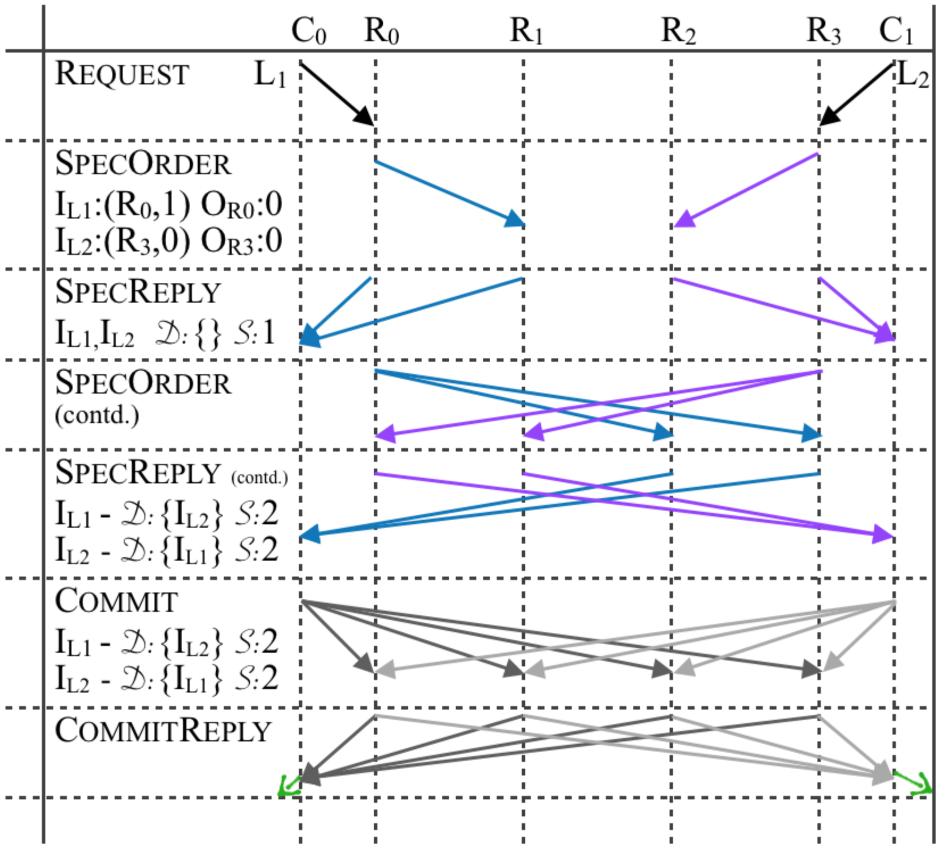


Figure 4.2: ezBFT: An example of a slow path execution.

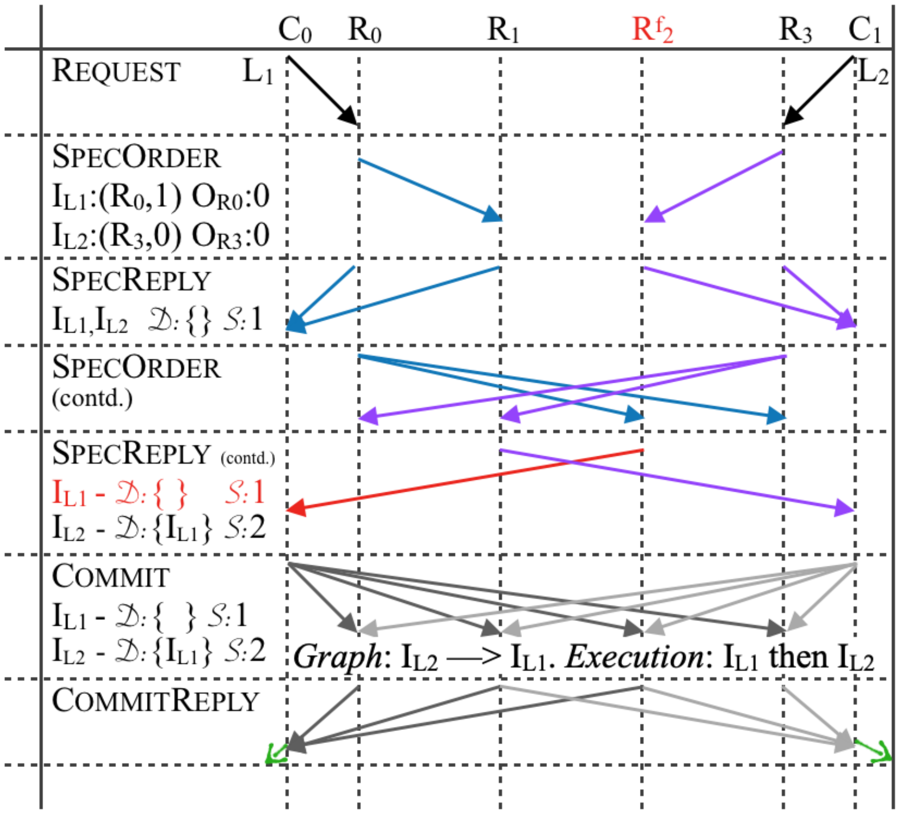


Figure 4.3: An example of a slow path with a faulty replica.

Example with a faulty replica

Figure 4.3 shows an example of the slow path that is very similar to that of Figure 4.2, but with a faulty replica R_2 that misbehaves. Notice that the Request and SpecOrder steps (first four rows) remain the same. Upon receipt of the SpecOrder message from R_0 and R_3 for L_1 and L_2, respectively, the replicas collect the dependency set and update the sequence number, and send back SpecReply messages to the client. For L_1, R_0 and R_1 send $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$; however, R_2 misbehaves and sends $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$, even though it received L_2 before L_1. For L_2, R_2 and R_3 send $\mathcal{D}'_{L_2} = \{\}$ and $\mathcal{S}'_{L_2} = 1$; R_1 sends $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$. c_0 receives a quorum of $2f + 1$ SpecReply messages, and sends a Commit message with an empty dependency set and a sequence number 1. On the other hand, R_1 that participated in command L_1's quorum sends back a correct dependency set and sequence number. Therefore, the final commit message for L_2 will have L_1 in its dependency set. Thus, even though replicas immediately execute L_1 since L_1's final dependency set is empty, they cannot do so for L_2. Correct replicas must wait until L_1 is *committed* before constructing the graph, at which point L_1 will be executed first because of the smallest sequence number, followed by L_2.

4.3.4 Triggering Owner Changes

ezBFT employs a mechanism at the clients to monitor the replicas and take actions to restore the service when progress is not being made. Although the slow path helps overcome the effects of a participant byzantine replica, it does ensure progress of a command when its command-leader, the replica that proposed that command, is byzantine. From the client-side, two events can be observed to identify misbehaving command-leaders.

4.3 The client times-out waiting for reply from the replicas.

After the client sends a request with command L, it starts another timer, in addition to the one for slow-path, waiting for responses. If the client receives zero or fewer than $2f + 1$ responses within the timeout, it sends the $\langle \text{Request}, L, t, c, R_i \rangle_{\sigma_c}$ message to all the replicas, where R_i is the original recipient of the message.

When replica R_j receives the message, it takes one of the following two actions. If the request matches or has a lower timestamp t than the currently cached timestamp for c, then the cached response is returned to c. Otherwise, the replica sends a $\langle \text{ResendReq}, m, R_j \rangle$ message where $m = \langle \text{Request}, L, t, c, R_i \rangle_{\sigma_c}$ to R_i and starts a timer. If the timer expires before the receipt of a SpecOrder message, R_j initiates an ownership change.

4.4 The client receives responses indicating inconsistent ordering by the command-leader and sends a proof of misbehavior to the replicas to initiate an ownership change for the command-leader's instance space.

Even though a client may receive prompt replies from the replicas, it must check for inconsistencies leading to a *proof of misbehavior* against the command-leader. The $\langle\langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t\rangle_{\sigma_{Ri}}, R_k, rep, SO\rangle$ messages from different replicas are said to match if they have identical O_{Ri}, I, $\mathcal{D}$, $\mathcal{S}$, c, t, and rep fields, but the contention may affect the equality of the dependency set and sequence number fields. Thus, the command-leader is said to misbehave if it sends SPECORDER messages with different instance numbers to different replicas (i.e., if the I field varies between the replicas). The client c can identify by inspecting SPECORDER SO message embedded in the SPECREPLY message received from the replicas. In this case, the client collects a pair of such messages to construct a $\langle \text{POM}, O_{Ri}, POM\rangle$ message, where POM is a pair of SPECREPLY messages, proving misbehavior by the command-leader R_i of L.

4.3.5 The Owner Change Protocol

An ownership change is triggered for an instance space if its original owner is *faulty*. However, to initiate an ownership change, there must exist either a proof of misbehavior against the owner, or enough replicas must have timed out waiting for a reply from the owner.

A replica R_j commits to an ownership change by sending a $\langle \text{STARTOWNERCHANGE}, R_i, O_{Ri}\rangle$ message to other replicas, where R_i is the suspected replica and O_{Ri} is its owner number.

When another replica R_k receives at least $f + 1$ STARTOWNERCHANGE messages for R_i, it commits to an ownership change. From this point forward, R_k will not participate in R_i's instance space. The new owner number is calculated as $O'_{Ri} = O_{Ri} + 1$, and the new command-leader is identified using O'_{Ri} mod N (henceforth R_l). Replicas that have committed to an owner-change send $\langle \text{OWNERCHANGE}\rangle$ messages to the new leader. Once the new command-leader R_l receives $2f + 1$ OWNERCHANGE messages, it becomes the new owner of R_i's instance space and finalizes its history.

Each replica sends an OWNERCHANGE message containing its view of R_i's instance space, i.e., the instances (speculative) executed or committed since the last checkpoint, and the commit-certificate with the highest owner number that it had previously responded to with a commit message, if any. The new owner collects a set P of OWNERCHANGE messages and selects only the one that satisfies one of the following conditions. For clarity, we label the sequence of instances in each OWNERCHANGE message as P_i, P_j, and so on.

There exists a sequence P_i that is the longest and satisfies one of the following conditions.

Condition 1 P_i has COMMIT messages with the highest owner number to prove its entries.

Condition 2 P_i has at least $f + 1$ SPECREPLY messages with the highest owner number to prove its entries.

If there exists a sequence P_j that extends a P_i satisfying any of the above conditions, then

P_j is a valid extension of P_i if one of the following conditions hold:

1. P_i satisfies Condition 1, and for every command L in P_j not in P_i, L has at least $f+1$ SpecReply messages with the same highest owner number as P_i.

2. P_i satisfies Condition 2, and for every command L in P_j not in P_i, L has a signed Commit message with the same highest owner number as P_i.

The new owner sends a NewOwner message to all the replicas. The message includes the new owner number O'_{R0}, the set P of OwnerChange messages that the owner collected as a proof, and the set of safe instances G produced using Condition 1 and Condition 2. A replica accepts a NewOwner message if it is valid, and applies the instances from G in R_i's instance space. If necessary, it rolls-back the speculatively executed requests and re-executes them again.

At this point, R_i's instance space is frozen. No new commands are ordered in the instance space, because each replica has its own instance space that it can use to propose its command. The owner change is used to ensure the safety of commands proposed by the faulty replicas.

4.3.6 Correctness

We formally specified ezBFT in TLA+ and model-checked using the TLC model checker. The TLA+ specification is provided in a technical report [21]. In this section, we provide an intuition of how ezBFT achieves its properties.

Nontriviality. Since clients must sign the requests they send, a malicious primary replica cannot modify them without being suspected. Thus, replicas only execute requests proposed by clients.

Consistency. To prove consistency, we need to show that if a replica R_j commits L at instance I, then for any replica R_k that commits L' at I, L and L' must be the same command.

To prove this, consider the following. If R_j commits L at $I = \langle R_i, - \rangle$, then an order change should have been executed for replica R_i's instance space. If R_j is correct, then it would have determined that L was executed at I using the commit certificate in the form of SpecOrder or Commit messages embedded within the ChangeOwner messages. Thus, L and L' must be the same. If R_j is malicious, then the correct replicas will detect it using the invalid progress-certificate received. This will cause an ownership change.

In addition, we need to also show that conflicting requests L and L' are committed and executed in the same order across all correct replicas. Assume that L commits with $\mathcal{D}$ and $\mathcal{S}$, while L' commits with $\mathcal{D}'$ and $\mathcal{S}'$. If L and L' conflict, then at least one correct replica must have responded to each other in the dependency set among a quorum of $2f+1$

replies received by the client. Thus, L will be in L''s dependency set and/or viceversa. The execution algorithm is deterministic, and it uses the sequence number to break ties. Thus, conflicting requests will be executed in the same order across all correct replicas.

Stability. Since only f replicas can be byzantine, there must exist at least $2f + 1$ replicas with the correct history. During an ownership change, $2f + 1$ replicas should send their history to the new owner which then validates it. Thus, if a request is committed at some instance, it will be extracted from history after any subsequent owner changes and committed at same instance at all correct replicas.

Liveness. Liveness is guaranteed as long as fewer than f replicas crash. Each primary replica attempts to take the fast path with a quorum of $3f + 1$ replicas. When faults occur and a quorum of $3f + 1$ replicas is not possible, the client pursues the slow path with $2f + 1$ replicas and terminates in two additional communication steps.

4.4 Evaluation

We implemented EZBFT, and its state-of-the-art competitors PBFT, FaB, and Zyzzyva in Go, version 1.10. In order to evaluate all systems in a common framework, we used `gRPC` [8] for communication and `protobuf` [44] for message serialization. We used the HMAC [54] and ECDSA [50] algorithms in Go's *crypto* package to authenticate the messages exchanged by the clients and the replicas. The systems were deployed in different Amazon Web Service (AWS) regions using the EC2 infrastructure [15]. The VM instance used was m4.2xlarge with 8 vCPUs and 32GB of memory, running Ubuntu 16.04. We implemented a replicated key-value store to evaluate the protocols. Note that, for Zyzzyva and EZBFT, the client process implements the logic for the client portion of the respective protocols.

Among the protocols evaluated, only EZBFT is affected by contention. Contention, in the context of a replicated key-value store, is defined as the percentage of requests that concurrently access the same key. Prior work [73] has shown that, in practice, contention is usually between 0% and 2%. Thus, a 2% contention means that roughly 2% of the requests issued by clients target the same key, and the remaining requests target clients' own (non-overlapping) set of keys. However, we evaluate EZBFT at higher contention levels for completeness.

4.4.1 Client-side Latency

To understand EZBFT's effectiveness in achieving optimal latency at each geographical region, we devised two experiments to measure the average latency experienced by clients located at each region for each of the protocols.

Experiment 1

We deployed the protocols with four replica nodes in the AWS regions: US-East-1 (Virginia), India, Australia, and Japan. At each node, we also co-located a client that sends requests to the replicas. For single primary-based protocols (PBFT, FaB, Zyzzyva), the primary was set to US-East replica; thus, clients in other replicas send their requests to the primary. For ezBFT, the client sends its requests to the nearest replica (which is in the same region). The clients send requests in closed-loop, meaning that a client will wait for a reply to its previous request before sending another one.

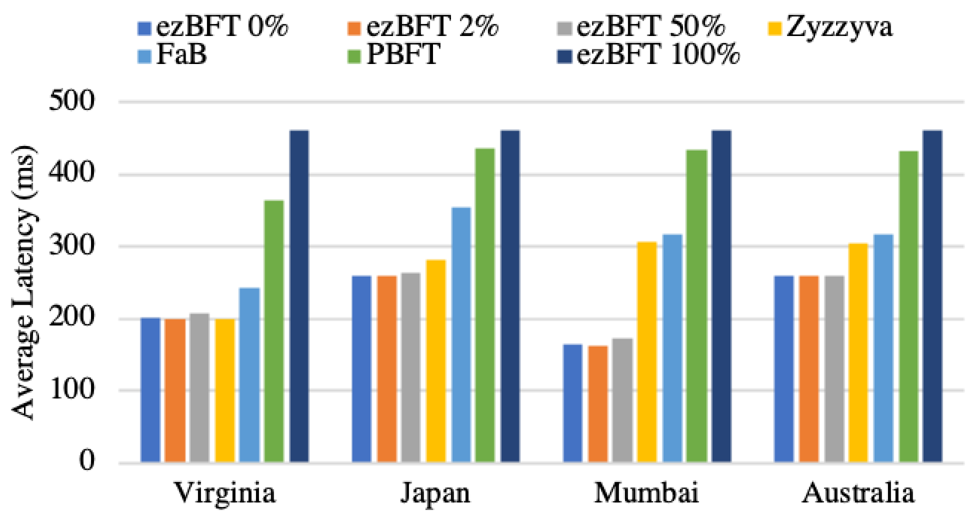
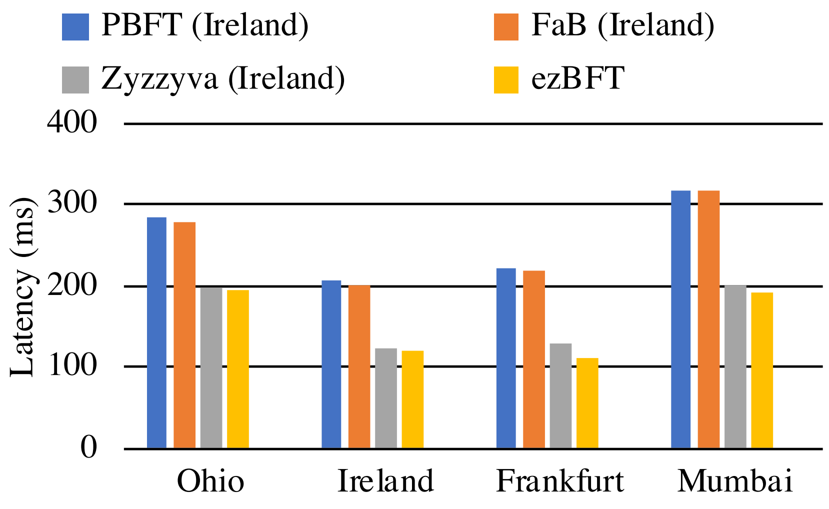


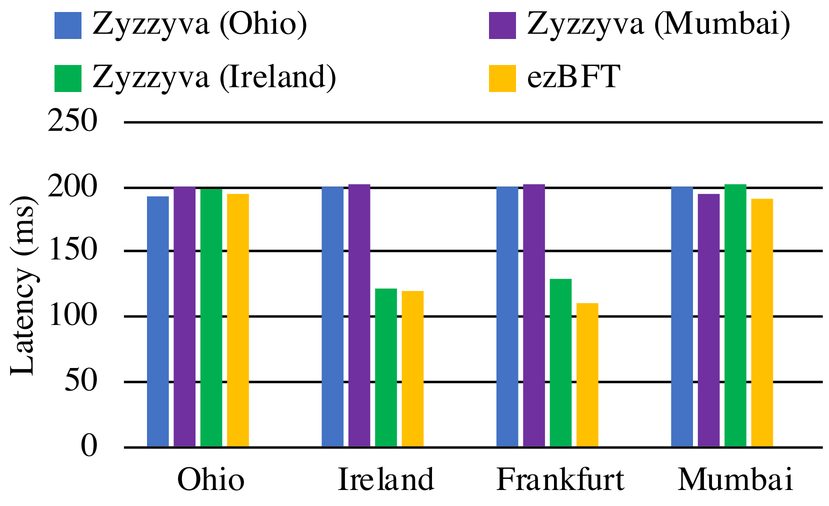
Figure 4.4: Average latencies for Experiment 1. All primaries are in US-East-1 region. The latency is shown per region as recorded by the clients in that region.

Figure 4.4 shows the average latency (in milliseconds) observed by the clients located in their respective regions (shown on x-axis) for each of the four protocols. For ezBFT, the latency was measured at different contention levels: 0%, 2%, 50%, and 100%; the suffix in the legend indicates the contention. Among primary-based protocols, PBFT suffers the highest latency, because it takes five communication steps to deliver a request. FaB performs better than PBFT with four communication steps, but Zyzzyva performs the best among primary-based protocols using only three communication steps. Overall, ezBFT performs as good as or better than Zyzzyva, for up to 50% contention. In the US-East-1 region, both Zyzzyva and ezBFT have about the same latency because they have the same number of communication steps and their primaries are located in the same region. However, for the remaining regions, Zyzzva clients must forward their requests to US-East-1, while ezBFT clients simply send their requests to their local replica, which orders them. At 100% contention, five communication steps required for total-order increases ezBFT's latency close to that of PBFT's.

Experiment 2



(a) Average latencies for Experiment 2.



(b) Average latencies for Experiment 2. Zyzzva's primary is at different regions.

Figure 4.5: Best case for Zyzzyva and the effect of moving primary to different regions. Experiments reveal EZBFT's effectiveness. Legend entries show primary's location in parenthesis.

To better understand Zyzzyva's best and worst-case performances and how they fare against EZBFT, we identified another set of AWS regions: US-East-2 (Ohio), Ireland, Frankfurt, and India. This experiment was run similar to that of Figure 4.4. The primary was placed in Ireland. The results are shown in Figure 4.5. Figure 4.5a shows the average latencies as observed by the clients in each deployed region for each of the four protocols. The choice of

Ireland as the primary region represents the best case for Zyzzyva. Hence, ezBFT performs very similar to Zyzzyva.

In Experiment 1, the regions mostly had non-overlapping paths between them, and thus the first communication step of sending the request to the leader can be seen clearly (notice Mumbai in Figure 4.4. On the other hand, in Experiment 2, connections between the regions have overlapping paths. For example, sending a request from Ohio to Mumbai for ezBFT will take about the same time as sending a request from Ohio to Mumbai via the primary at Ireland for Zyzzyva.

Figure 4.5b shows the effect of moving the primary to different regions. We disregard PBFT and FaB in this case, as their performance do not improve. For Zyzzyva, moving the primary to US-East-2 or India substantially increases its overall latency. In such cases, ezBFT's latency is up to 45% lower than Zyzzyva's. This data-point is particularly important as it reveals how the primary's placement affects the latency.

To curb the negative effects of byzantine primary replicas, in [31], the authors propose to frequently move the primary (this strategy is adopted by other protocols including Zyzzyva). From Figure 4.5b, we can extrapolate that such frequent movements can negatively impact latencies over time. Given these challenges, we argue that ezBFT's leaderless nature is a better fit for geo-scale deployments.

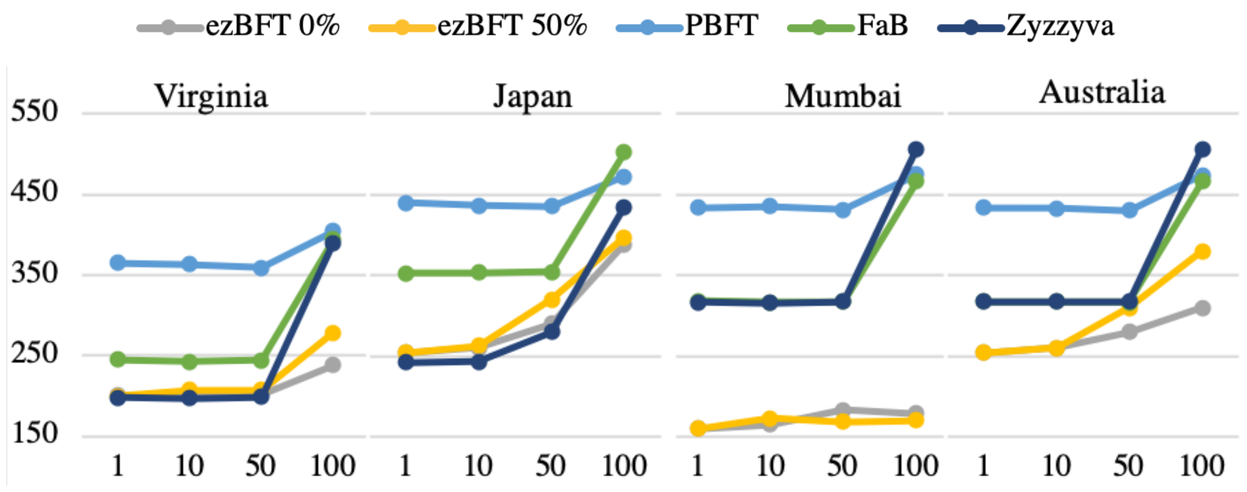


Figure 4.6: Latency per location while varying the number of connected clients (1 – 100) per region.

4.4.2 Client Scalability

Another important aspect of ezBFT is its ability to maintain low client-side latency even as the number of connected clients increases. For this experiment, we deployed the protocols in Virginia, Japan, Mumbai, and Australia, and measured client-side latency per region by varying the number of connected clients. Figure 4.6 shows the results. Notice that as Zyzzyva approaches 100 connected clients per region, it suffers from an exponential increase in latency. However, EZBFT, even at 50% contention, is able to scale better with the number of clients. Particularly, in Mumbai, EZBFT maintains a stable latency even at 100 clients per region, while Zyzzyva's latency shoots up.

4.4.3 Server-side Throughput

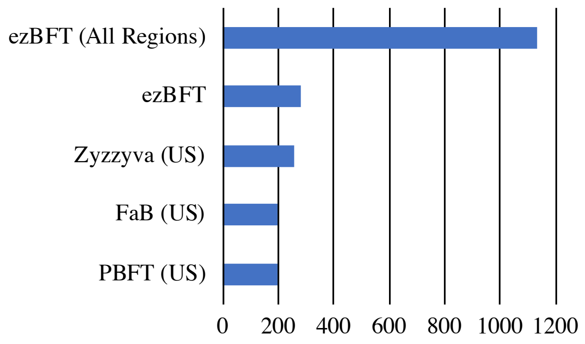


Figure 4.7: Throughput of EZBFT and competitor BFT protocols.

We also measured the peak throughput of the protocols. For this experiment, we deployed the protocols in five AWS regions: US-East-1 (Virginia), India, Australia, and Japan. We co-located ten clients with the primary replica in US-East-1. Unlike the experiments so far, here the clients send requests in an open-loop, meaning that they continuously and asynchronously send requests before receiving replies. The requests consists of an 8-byte key and a 16-byte value.

Figure 4.7 shows the results. For EZBFT, we carried out two experiments: i) clients are placed only at US-East-1 (labelled EZBFT in the figure); and ii) clients are placed at every region (labelled "EZBFT (All Regions)" in the figure). Due to the leaderless characteristic, each of the replicas can feed requests into the system, increasing the overall throughput. The contention was set to 0%, and no batching was done.

Observe that when clients are placed only at US-East-1, EZBFT performs at par or slightly better than others. On the other hand, when clients are placed in all the other regions, which does not yield any benefit for other protocols, EZBFT's throughput increases by as much as four times, as all EZBFT replicas are able to process and deliver requests concurrently.

Chapter 5

Dester

This chapter proposes Dester, a leaderless hybrid fault-tolerant state machine replication protocol built *ground-up* for achieving high performance in the geo-scale environments. The leaderless nature allows every replica to process client commands by only relying on a set of closest replicas, thus providing low client-side latencies and high system throughput. Dester is, in part, made possible by a novel trusted subsystem called TruDep that provides the necessary trust required to ensure secure, leaderless operation. TruDep was designed specifically for trusted execution environments (such as Intel SGX) with a goal of minimizing the amount of trusted code, thus nullifying the possibility of bugs in the subsystem.

The rest of this chapter is organized as follows. Section 5.1 highlights the benefits of Dester and the innovations behind it. Section 5.2 describes the interfaces and implementation of the TruDep subsystem. Section 5.3 describes Dester algorithm in detail with the help of some definitions. Finally, Section 5.4 evaluates the protocol.

5.1 Overview

Byzantine Fault-Tolerant (BFT) SMR solutions (e.g., [12, 16, 27, 30, 33, 34, 38, 45, 52]) can defend applications from arbitrary faults. However, such solutions are seldom used in practice due to their high costs – they require $3f+1$ nodes to tolerate f arbitrary failures. For many replicated systems, such as those maintained by a single enterprise-class organization within a private network, malicious attacks, where the adversary can take control of faulty machines as well as the network in a coordinated way, are essentially non-existent [64].

In that regard, Dester has been designed specifically to take advantage of the recent hardware and cloud trends – such as trusted execution environments and geo-scale deployments – to achieve a high performance system with an optimal replication factor and a practically sufficient fault tolerance model. The hybrid fault model adopted by Dester can defend systems from most byzantine behaviors, such as software bugs and hardware errors, as long as the trusted subsystem is uncompromised. Moreover, a substantial reduction in the number of nodes to $2f + 1$ (from $3f + 1$ in the BFT model) and the use of CPU-based trusted execution environments instead of external devices [32, 51, 85], help boost performance to a large extent [26].

Furthermore, Dester's leaderless operation is key in unlocking high performance in geo-scale deployments. Leaderless operation enables every replicas to propose and commit commands without relying on designated replicas. As we show in our evaluation in Section 5.4, the use of leader, or *primary* replicas has many disadvantages. Firstly, clients from non-leader sites experience very high latencies by forwarding commands to a (distant) leader. Secondly, very high load at the leader causes resource bottlenecks leading to low throughput. Finally, unavailability is inevitable under a faulty leader.

Dester exploits the notion of *command interference* to allow concurrent ordering and execution of commands originating at different replicas. The technique, adopted from the Generalized Consensus [56] definition in the CFT model, involves executing concurrent commands in any order as long as they do not interfere. This enables Dester replicas to commit client commands in two communication steps from receipt. Furthermore, Dester is cautious when ordering interfering commands concurrently, thus an additional communication step may be required depending on the outcome of prior steps.

To allow for a leaderless behavior even in cases of interfering commands, Dester collects command dependencies. Two commands are dependent, if they interfere and thus, a deterministic execution order is required to ensure system consistency. Replicas only execute a command, when at least a quorum of replicas observe or agree to the same set of dependencies for that command. Furthermore, the execution of command involves respecting these dependencies deterministically across replicas.

Dester's hybrid fault model is made possible by its trusted subsystem, TruDep. The subsystem has been designed from specifically to accommodate Dester's requirements and take unique advantage of flexibility provided by CPU-based TEEs. The purpose of the TruDep subsystem is to ensure that the messages exchanged by the Dester are correct and that malicious replicas cannot equivocate without detection. by taking advantage of standard library data structures such as maps.

5.2 The TruDep Subsystem

The role of a trusted component in a hybrid fault tolerance system is to ensure that replicas cannot equivocate without detection. In Dester, equivocation means that a replica can propose different commands to different replicas for the same ordering position or may withhold dependency information for commands from other replicas. TruDep's purpose is to prevent such equivocation by certifying messages sent by and verifying the messages received by replicas.

Figure 5.1 provides a summary of Dester's interfaces and their implementation details. Each replica has an instantiation of TruDep. The subsystem provides three interfaces: two for creating certificates and one for verifying. Every message a replica sends must be accompanied by a Dester certificate that other correct replicas can verify using their

TruDep Instance Variables:

`Key`	shared secret key
`counters`	an array of counters
`deps`	an array of maps of (string, integer) pairs
`tssID`	ID of TruDep instance

***CreateCertificate*:**

Arguments

`cID`	counter ID
`tv'`	next counter value
`deps'`	map of (key, instance) pairs
`m`	message

Implementation

If `!tv` then `tv = counters[cID]`

If `tv' > counters[cID]` and `tv == counters[cID]` and `valid(deps')`:

- set `counters[cID]` value to `tv'`
- generate HMAC using (`tssID, cID, tv', deps', tv, m`) and `secret`.

VerifyCertificate:

Arguments

`cID`	counter ID
`tv',tv`	next counter and optional current counter
value	
`deps'`	map of (key, instance) pairs
`m`	message
`hmac`	received hmac for message

Implementation

If `valid(deps')`:

- update `deps` with values from `deps'`
- `hmac'` = generate HMAC using (`tssID, cID, tv', deps', tv, m`) and `secret` with the appropriate tag.
- return `hmac == hmac'`

Figure 5.1: Summary of the TRUDEP Subsystem.

subsystem. We assume that the subsystem is instantiated by a trusted administrator with the secret keys and the unique identifier, `tssID`.

Every TRUDEP instance has the following variables, in addition to the secret key and identifier used for creating certificates. `counters` is an array of numbers, one for each replica in the system. A counter specifies the ordering for commands and is used to ensure that different commands are not proposed at the same ordering position. `deps` is an array of maps, one for each replica. Each map holds information about command interference information from another replica. The map's key is the command interference identifier, such as key in a key-value store (see Section 5.4), and the value is the highest counter value among the set of interfering commands.

Prior to certificate creation, the counter value and dependencies are validated and updated. The new counter value must be greater than previous value and the dependencies (`deps'`) must compare to `deps` at all replica indices. If so, the current replica's counter field is updated and the dependencies `deps` is updated with interference information about new command. Then, a certificate is created using the HMAC algorithm and returned.

During verification, the certificate is first validated and then the `deps` is updated. Only the remote replica's entries are updated, since TRUDEP only uses first hand information from the respective "owners" to ensure safety.

5.3 Protocol Design

In this section, we present DESTER in detail, along with an informal proof of its properties. The system is composed of two subprotocols: the commit protocol and the execution protocol. The commit protocol finds an ordering for client commands, while the execution protocol executes them in that committed order. We should note that the commit protocol does not produce a sequential order but a directed graph describing the relationship between commands. The execution protocol respects these constraints during execution.

The commit protocol works in two phases: fast and slow. A client command is first tried to be committed in the *fast phase*, and upon failure is finalized in the *slow phase*. A fast phase for a command is achievable under low contention and when there are at least a fast quorum of synchronously communicating replicas. Otherwise, an additional slow phase is required to finalize the command. The fast phase takes two communication steps, while the slow phase requires an additional step. Note that the client-replica message exchanges take two more communication steps.

5.3.1 Definitions

- **Command Interference.** A command encapsulates an operation that must be executed on the shared state. We say that two commands L_0 and L_1 are interfering if the execution of these commands in different orders on a given state will result in two final states. That is, if there exists a sequence of commands Σ such that the serial execution of Σ, L_0, L_1 is not equivalent to Σ, L_1, L_0, then L_0 and L_1 are interfering.

- **Instance space.** Instance space is a sequence of numbered slots to which client-commands are associated with. The sequence defines the execution order of requests, and the role of a consensus protocol is to reach agreement among a set of replicas on a common order. In Dester, every replica owns an instance space and mirrors other replicas' instance spaces. Replicas can only propose commands to its own space.

- **Instance Number.** An instance number, denoted I, is a tuple of the instance space (or replica) identifier and a slot identifier.

- **View.** The view number v is used to identify the current owner of an instance. This number becomes useful when a replica is faulty, and its instances must be recovered by other replicas. When replicas fail, another correct replica steps up to take control of the faulty replica's instance space. The leader of a replica R_0's instances can be identified from the view number using the formula $v_I \bmod N$, where I is the recovering instance number and N is the number of replicas. Initially, the view number of each instance is set to its replica's identifier (e.g., $v(I_{R0}) = 0$, $v(I_{R1}) = 1$, and so on).

- **Dependencies.** Due to the use of per-replica instance spaces, the protocol must agree on the relationship between the slots of different instances spaces. Dester does this via dependency collection, which uses the command interference relation. The dependency set $\mathcal{D}$ for command L is every other command L' that interferes with L.

- **Sequence Number ($\mathcal{S}$).** is a globally shared number that is used to break cycles in dependencies. It starts from one and is always set to be larger than all the sequence numbers of the interfering commands. The replica identifiers are used to break ties, when interfering commands originating from different command-leaders are assigned the same sequence number due to concurrency.

Dester uses two kinds of quorums: a fast quorum with $f + \left\lfloor \frac{f+1}{2} \right\rfloor$ replicas, and a slow quorum with $f + 1$ replicas. A fast quorum is required for guaranteeing responses in two communication steps under Dester's leaderless operation. Slow quorum is used for the third communication step and during the recovery procedure.

Protocol Properties. Dester ensures the following:

1. **Nontriviality.** Any request committed and executed by a replica must have been proposed by a client.

2. **Stability.** For any replica, the set of committed requests at any time is a subset of the committed requests at any later time. If at time t_1, a replica R_i has a request L committed at some instance I_L, then R_i will have L committed in I_L at any later time $t_2 > t_1$.

3. **Consistency.** Two replicas can never have different requests committed for the same instance.

4. **Liveness.** Requests will eventually be committed by every non-faulty replica, as long as at least $2f + 1$ replicas are correct.

5.3.2 The Fast Phase

A command, consisting of the operation to be performed on the replicated state, originates at a client, and is sent to a (preferably closer) replica. Every command is signed with the client's private key to maintain its integrity. The replica receives the command and starts the fast phase. It works as follows:

1. A replica receives a client command, assigns an instance number, computes dependencies and a sequence number, and forwards to other replicas.

When a replica R_i receives a client command L from client c, it takes ownership of that command. R_i assigned the next available instance number in R_i's instance space. Further, it produces a dependency set $\mathcal{D}$ and a sequence number $\mathcal{S}$ as described in previous section.

The command and its attributes are sent to other replicas in a $\langle\langle \text{PROPOSE}, v, L, I_L, \mathcal{D}, \mathcal{S}\rangle, \mathcal{A}\rangle$ message, where v is the view number, I_L is L's instance number and $\mathcal{A}$ is the certificate produced by calling the CREATECERTIFICATE method of R_i's TRUDEP instance. The counter value is obtained by shifting v to the higher order bits and then ORing with the instance number. This way both the attributes can be expressed with a single counter. Furthermore, when changing views (see Section 5.3.5), it is possible to generate certificates for the new view but still retain the instance number.

2. Other replicas receive and verify the PROPOSE message, add L and its attributes to its log, compute $\mathcal{D}'$ and $\mathcal{S}'$ over their local state, and send a COMMIT message.

When replica R_j receives a message $\langle\langle \text{PROPOSE}, v, L, I_L, \mathcal{D}, \mathcal{S}\rangle, \mathcal{A}\rangle$ from replica R_i, it calls VERIFYCERTIFICATE method of R_j's TRUDEP to verify the message's integrity and ensures that L is a valid command and that the instance number I_L is next available and has no gaps. R_j updates the dependencies and sequence number from the message to its local log and produce new sets of dependencies $\mathcal{D}'$ and sequence number $\mathcal{S}$' over its local log. It, then, broadcasts a $\langle\langle \text{COMMIT}, v, L, I_L, \mathcal{D}', \mathcal{S}'\rangle, \mathcal{A}\rangle$ message to other replicas, where $\mathcal{A}$ is certificate produced by R_j for L's COMMIT message.

3. Replicas receive either a fast or slow quorum of Commit messages.

The number of Commit messages plus the original Propose message, and their attributes determine if a command can be committed in the fast phase or whether a slow phase is necessary. Furthermore, every replica R_i specifies a set of replicas that every other replica must use as the fast and slow quorums for commands proposed by R_i. This is to ensure that every replica uses the same deterministic quorum to make subsequent decisions.

3.1 Replicas receive a fast quorum of Commit messages with matching $\mathcal{D}$ and $\mathcal{S}$ fields.

If the fast quorum of Commit messages have matching $\mathcal{D}$ and $\mathcal{S}$ fields, the command L is committed in the fast phase and is enqueued for execution (see Section 5.3.3). Note that the original Propose message from the command-leader serves as its Commit message.

5.3.3 Execution Protocol

Dester's execution protocol is responsible for deciphering the final execution order of committed commands using their attributes and executing them in that order. One of the fundamental differences between Dester and existing hybrid solutions rely in the execution protocol, because existing hybrid solutions use primary replicas to commit commands sequentially, while Dester allows commands to be committed concurrently and independently by replicas. Thus, Dester must ensure that command dependencies are respected during execution and that the execution is deterministic across replicas. This is the responsibility execution subprotocol and it works as follows:

Each replica takes the following steps:

1. Waits for the command to be enqueued for execution.
2. A dependency graph is constructed by including R and all its dependencies in $\mathcal{D}$ as nodes and adding edges between nodes indicating the dependencies. The procedure is repeated recursively for each dependency.
3. Strongly connected components are identified and sorted topologically.
4. Starting from the inverse topological order, every strongly connected component is identified, and all the requests within the component are sorted in the sequence number order. The requests are then executed in the sequence number order, breaking ties using replica identifiers.

Once the command is executed, a reply is sent back to the client. The client required at least $f + 1$ responses with identical values to validate correct (non-byzantine) behavior by replicas.

5.3.4 The Slow Phase Protocol

The lack of a fast quorum of COMMIT messages with matching $\mathcal{D}$ and $\mathcal{S}$ fields implies that many of the replicas in the quorum have different perspective of dependencies for command L likely caused due to concurrent conflicting proposals. Thus, to decide the final set of dependencies and sequence number, a slow phase protocol, as described below, is necessary.

Note that non-matching attribute fields do not imply malicious behavior by replicas, since the TRUDEP subsystem ensures that only valid dependencies are included in the messages.

3.2 Replicas receive at least a slow quorum of COMMIT messages with non-matching fields.

Upon receipt of the PROPOSE message, replicas that were part of L's fast quorum set a timer and wait for the receipt of COMMIT messages. If, when the timeout occurs, the responses are fewer than a fast quorum, or contain potentially non-matching dependency sets, replicas construct the final dependency set and sequence number for L using a slow quorum of commit messages. Similar to the fast quorums, the command-leader R_i for L also enforces a fixed slow quorum for its commands that all replicas must use for R_i's commands. This is to ensure that replicas produce identical, deterministic set of final dependencies and sequence number, which is required for the success of the slow phase.

The final dependency set $\mathcal{D}''$ and sequence number $\mathcal{S}''$ is computed by taking the union of dependency sets and the maximum of sequence numbers, respectively. The fast quorum replicas for L broadcast a $\langle \text{FINALIZE}, v, L, I_L, \mathcal{D}'', \mathcal{S}'', m \rangle$ message, where m is a set of COMMIT messages proving the final dependencies. Unlike fast phase messages, the FINALIZE message is not certified by TRUDEP.

3.3 Replicas receive a slow quorum of FINALIZE message and commit the command

Upon receipt of a slow quorum of FINALIZE messages, replicas verify that each received message has a valid proof of COMMIT messages supporting their final dependency set and sequence number. Then, replicas update their local logs with the final attributes from any of the messages. The command L is marked *committed* and queued for execution.

Example. Figure 5.2 shows an example execution of DESTER, highlighting both fast phase and slow phase commits. The example consists of three replicas R_0 through R_2 and two clients. Since $N = 3$ replicas, both the fast quorum and slow quorum consist of two replicas each.

① Two clients c_0 and c_1 send signed REQUEST messages to replicas R_0 and R_2, respectively, to execute commands L_1 and L_2, respectively, on the replicated service. Assume that L_1 and L_2 conflict. ② Replica R_0 assigns the lowest available instance number of $\langle R_0, 0 \rangle$ to L_1. Thus, R_0 collects a dependency set $\mathcal{D}_{L_1} = \{\}$ and assigns a sequence number $\mathcal{S}_{L_1} = 1$ to L_1.

Then, R_0 sends a PROPOSE message with its commands and metadata to other replicas. The message also includes a certification produced by R_0's TRUDEP instance. (3) R_1 receives the proposal, verifies it using its TRUDEP instance, and computes a dependency set and sequence number that is unchanged from the one received. It sends a COMMIT message with the information to other replicas. R_1 certifies the message using its trusted subsystem before broadcasting. Since the PROPOSE message from the command-leader also serves as the commit, R_2 starts executing L_1. When R_1 receives the COMMIT message, it will start executing it as well. Since, they do not have any dependencies, they can be immediately executed and a response can be sent to the client. This concludes the fast phase decision of L_1 in two communication steps (excluding client messages). Note that, in this example, we assume that R_2 is unaware of command L_1 since it did not receive any PROPOSE or COMMIT messages for L_1.

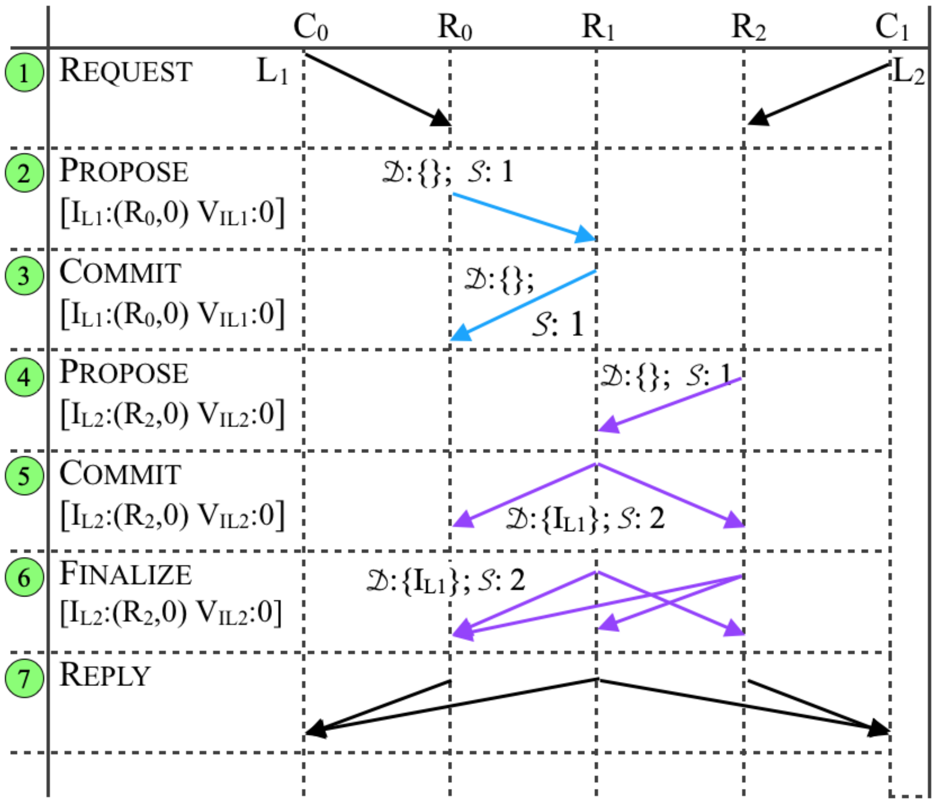

C0
R0
R1
R2
C1
1 REQUEST
L1
L2
2 PROPOSE [IL1:(R0,0) VIL1:0]
𝔇:{}; 𝒮: 1
3 COMMIT [IL1:(R0,0) VIL1:0]
𝔇:{}; 𝒮: 1
4 PROPOSE [IL2:(R2,0) VIL2:0]
𝔇:{}; 𝒮: 1
5 COMMIT [IL2:(R2,0) VIL2:0]
𝔇:{IL1}; 𝒮: 2
6 FINALIZE [IL2:(R2,0) VIL2:0]
𝔇:{IL1}; 𝒮: 2
7 REPLY

Figure 5.2: DESTER: An example execution.

(4) R_2 assigns the instance number $\langle R_3, 0\rangle$ to L_1; the computed dependency set is $\mathcal{D}_{L_2} = \{\}$, and sequence number is $\mathcal{S}_{L_2} = 1$. R_2 sends a certified PROPOSE message to R_1. When R_1 receives and verifies the proposal, it updates its log and then computes the new set of dependencies and sequence number. Since L_2 arrived after L_1 at R_1 and since the two commands conflict, R_1 must include L_1 in L_2's dependency set, and thus it should also propose a higher sequence number $\mathcal{S} = 2$. (5) R_1 sends a COMMIT message with $\mathcal{D}_{L_2} = \{L_1\}$

and $\mathcal{S}_{L_2} = 2$. Even though there is a fast quorum of two responses, L_2 cannot be committed in the fast phase, since the dependency snapshot vary between the L_2's quorum replicas R_2 and R_3. This marks the beginning of the slow phase.

⑥ For both R_1 and R_2, the PROPOSE message and a COMMIT message is enough to constitute a slow quorum. Thus, when both R_1 and R_2 observe a slow quorum of non-matching dependencies, they accumulate the dependency and choose a maximum sequence number and send a FINALIZE message to all replicas. Note that unlike PROPOSE and COMMIT messages, the FINALIZE message need not be certified by TRUDEP, but requires a proof of a quorum of COMMIT (possibly including the original PROPOSE message) to be accepted by other replicas. Once a slow quorum of valid FINALIZE messages arrive at replicas, they can begin execution the commands. Since L_1 is in L_2's dependency set and has a lower sequence number, L_2 should only execute after L_1. ⑦ Once executed, the reply is sent back to the respective clients.

5.3.5 Recovery

A client command is committed using either the fast or slow phase protocol under normal operating conditions. However, commands may not be committed when replicas act maliciously and do not exchange messages in a timely manner. The TRUDEP subsystem only ensures that replicas cannot behave maliciously without being detected, but DESTER, like any BFT protocol, cannot not identify between malicious and crashed or unreachable replicas. Thus, a malicious replica may simply cease to send protocol messages in a timely manner. This can happen at two scenarios in the DESTER's commit protocol. Firstly, in DESTER, command-leader establishes quorum replicas for subsequent message rounds, but those replicas may not send timely messages ceasing progress. Secondly, the command-leader may stop sending proposals. The recovery protocol helps with coping with both these scenarios. The recovery procedure starts at the client.

After the client sends a request with command L, it starts a timer and waits for responses. If the client receives fewer than $f+1$ responses within the timeout, it forwards the command to all the replicas indicating that the message was originally sent to R_i. Any replica R_j that received this forwarded command returns the cached response to the client, if already executed. Otherwise, R_j will forward the request to the command-leader and expect a proposal within a time period. The command-leader is said to misbehave when the period elapses and no proposal has been received. Thus, command L must be recovered.

If the command-leader re-sends a proposal, but the command could not be committed, then it is likely due to a faulty replica in the predetermined quorums. The command-leader cannot adjust the composition of quorum because some replicas may have committed and executed the command already using the previous quorum. Therefore, to commit L at all replicas, it must be recovered. When replica R_j time-out waiting for the commitment of forwarded command L, it starts the recovery procedure for L by broadcasting a $\langle \text{RECOVER}, v', I_L \rangle$, where I_L is the instance number to be recovered and v' is the new view number calculated as $v+1$, where v is the previous view number.

Other replicas send a RecoverR message in response to the Recover message to the new leader of L as indicated by the new view v'. The reply message consists of the $\mathcal{D}$ and $\mathcal{S}$ attributes of command L along with any Prepare or Commit messages previously exchanged by that replica. When the new leader R_j receives a slow quorum of RecoverR messages, it proceeds with the following steps.

1. If at least one replica committed L at I_L with a $\mathcal{D}$ and $\mathcal{S}$ that is supported with a fast quorum of Commit or a slow quorum of Finalize messages, R_j sends a Fix message with the new view number and the attributes received.

2. If at least one replica has received a slow quorum of commit messages, then the command can be commit by sending a Fix messages including a slow quorum of Commit messages as proof.

When a replica receives a Fix message from the leader, it applies the command in its log and prepares for its execution. Furthermore, the Fix message is certified using TruDep using the new view number v' and instance number I_L. Note that since the previous counter value is of the form $[v|I_L]$ for the faulty replica, by creating a certificate for v', we will simply update the counter to $[v'|I_L]$ and it will still satisfy the monotonically increasing property of the counter.

5.3.6 Correctness

In this section, we provide an intuition of how Dester achieves its properties.

Nontriviality. Since clients must sign the requests they send, a malicious primary replica cannot modify them without being suspected. Thus, replicas only execute requests proposed by clients.

Consistency. To prove consistency, we need to show that if a replica R_j commits L at instance I, then for any replica R_k that commits L' at I, L and L' must be the same command.

To prove this, consider the following. If R_j commits L at $I = \langle R_i, - \rangle$, then an order change should have been executed for replica R_i's instance space. If R_j is correct, then it would have determined that L was executed at I using the commit certificate in the form of Propose or Commit messages embedded within the recovery messages. Thus, L and L' must be the same. Even if R_j is malicious, it cannot equivocate thanks to TruDep. R_j may go silent, but this will cause an leader change.

In addition, we need to also show that conflicting requests L and L' are committed and executed in the same order across all correct replicas. Assume that L commits with $\mathcal{D}$ and $\mathcal{S}$, while L' commits with $\mathcal{D}'$ and $\mathcal{S}'$. If L and L' conflict, then at least one correct replica must have responded to each other in the dependency set among a quorum of replies received by the client. Thus, L will be in L''s dependency set and/or viceversa. The execution algorithm is deterministic, and it uses the sequence number to break ties. Thus, conflicting requests will be executed in the same order across all correct replicas.

Stability. Since only f replicas can be byzantine, there must exist at least $f + 1$ replicas with the correct history. During an leader change, $f + 1$ replicas should send the history of recovering instances to the new leader which then validates it. Thus, if a recovering instance has been committed previously, it will be extracted from the history after any subsequent leader changes and committed at same instance at all correct replicas.

Liveness. Liveness is guaranteed as long as fewer than f replicas crash. Each primary replica attempts to take the fast phase with a fast quorum of replicas. When fast phase failed, the client pursues the slow phase with a slow quorum of replicas and terminates with one additional communication steps.

5.4 Evaluation

We implemented DESTER, and its state-of-the-art competitors Hybster [26] and PBFT [30] in Go, version 1.11. Our trusted subsystem TRUDEP and Hybster's trusted counter TrInx were implemented in C using Intel SGX SDK and interfaced with the Go application using `cgo` [43]. In order to evaluate all systems in a common framework, we used `gRPC` [8] for communication and `protobuf` [44] for message serialization. We used the ECDSA [50] algorithms in Go's *crypto* package to authenticate the messages exchanged by the clients and the replicas.

Among the protocols evaluated, only DESTER is affected by contention. Contention, in the context of a replicated key-value store, is defined as the percentage of requests that concurrently access the same key. Prior work [73] has shown that, in practice, contention is usually between 0% and 2%. Thus, a 2% contention means that roughly 2% of the requests issued by clients target the same key, and the remaining requests target clients' own (non-overlapping) set of keys. However, we evaluate DESTER at higher contention levels for completeness.

Deployment. The systems were deployed on three SGX-capable virtual machines, each in different geographical region using the Microsoft Azure Cloud Platform [2]. Each VM belong to the DC2-series and has a 3.7GHz Intel Xeon E2176G Processor with 8GB RAM. The OS used was Ubuntu 16.04 with Intel SGX Driver v2.3.1 installed. Note that at the time of this writing, Azure's SGX-capable DC2 virtual machines were only available in two regions: East US and West Europe, but to witness the benefits of DESTER, at least three regions are necessary. Therefore, we created the third instance in East US, but proxied its incoming traffic via North Central region. Thus, the traffic to third instance will arrive at a (non-SGX) VM in the North Central US region, before being redirected back to the VM in East US. The outgoing traffic was not proxied. Thus, we refer to the location of the third VM as North Central US. Furthermore, the fourth replica for PBFT was created in the West Europe data center.

A note on other state-of-the-art systems. Our evaluation only focuses on comparison against software-based hybrid system, Hybster, and the most popular byzantine fault-tolerant system, PBFT. We do not consider previous hybrid proposals based on FPGAs and ASICs such as TrInc [61], CheapBFT [51], and MinBFT [85] as their hardware trusted components have higher overheads than CPU-based TEEs and thus perform poorly. Thus, we compare DESTER against Hybster. Byzantine fault tolerance is usually synonymous with PBFT, thus we show how DESTER fares against it.

5.4.1 Client-side Latency

To understand DESTER's effectiveness in achieving optimal latency at each geographical region, we devised an experiment to measure the average latency experienced by clients located at each region for each of the protocols. At each node, we also co-located a client that sends requests to the replicas. For leader-based protocols (Hybster and PBFT), the leader was set to East US replica; thus, clients in other regions send their requests to the leader. For DESTER, the client sends its requests to the nearest replica (which is in the same region). The clients send requests in closed-loop, meaning that a client will wait for a reply to its previous request before sending another one.

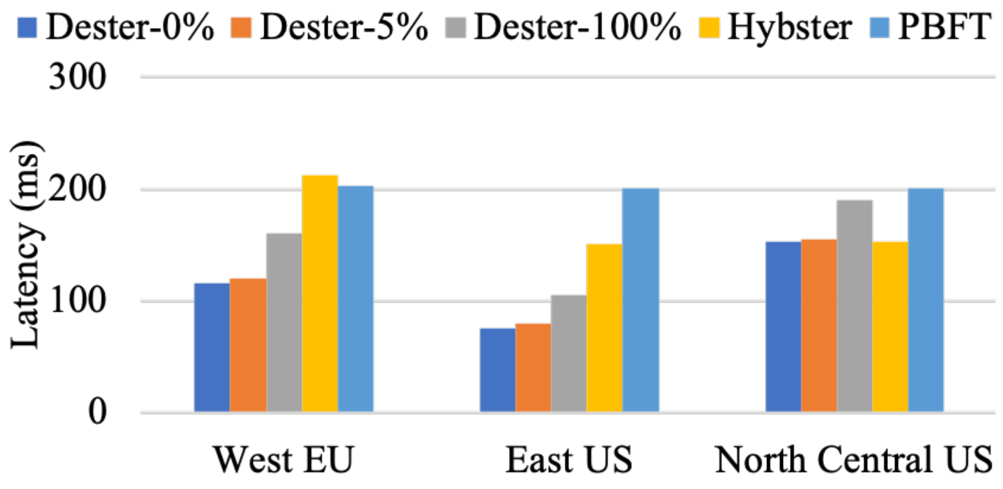


Figure 5.3: Average latencies for Experiment 1. All primaries are in North Central US region. The latency is shown per region as recorded by the clients in that region.

Figure 5.3 shows the average latency (in milliseconds) observed by the clients located in their respective regions (shown on x-axis) for each of the protocols. For DESTER, the latency was measured at different contention levels: 0%, 5%, and 100%; the suffix in the legend indicates the contention. Among leader-based protocols, PBFT suffers the highest latency, by contacting an additional faraway replica and taking three communications steps. Hybster only provides optimal latency at leader-site. DESTER in contrast is able to provide optimal latencies at every site, thus achieving huge latency savings of up to 50%.

5.4.2 Server-side Throughput

We also measured the peak throughput of the protocols. For this experiment, we co-located fifty clients with each replicas. The requests consists of a 16-byte key and a 1KB value. The contention was set to 5%, and no batching was done.

Figure 5.4 shows the results. PBFT provides the least throughput because of larger quorum sizes. Hybster is able to perform better than PBFT thanks to its smaller quorum and the user of CPU-based trusted execution environment. DESTER's leaderless characteristic enables each of the replicas to feed requests into the system, thus increasing the overall throughput by as much as 40% compared to Hybster.

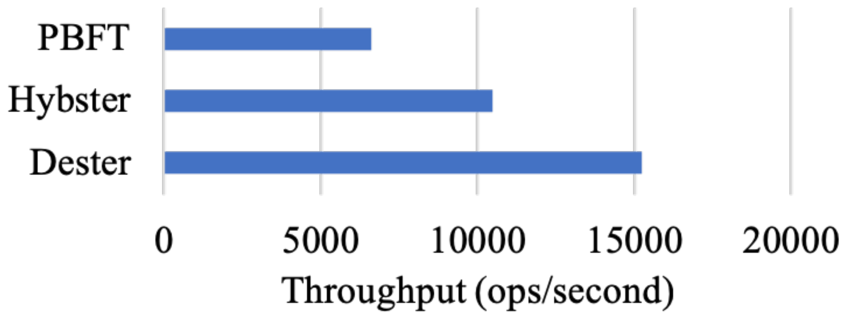


Figure 5.4: Throughput of DESTER and competitor BFT protocols.

5.4.3 Fault Tolerance

One of the crucial aspects of DESTER is its ability to provide better availability guarantees than leader-based systems. Unavailability is inevitable when leader replicas are faulty. To show this, we devised an experiment in which we terminate the leader replica in Hybster and a random replica in DESTER, about 20 seconds through the experiment. The results are in Figure 5.5. In DESTER, only clients from the crashed region timeout and reconnect to another region, while Hybster starts a leader change phase. Thus, there is only slight decrease

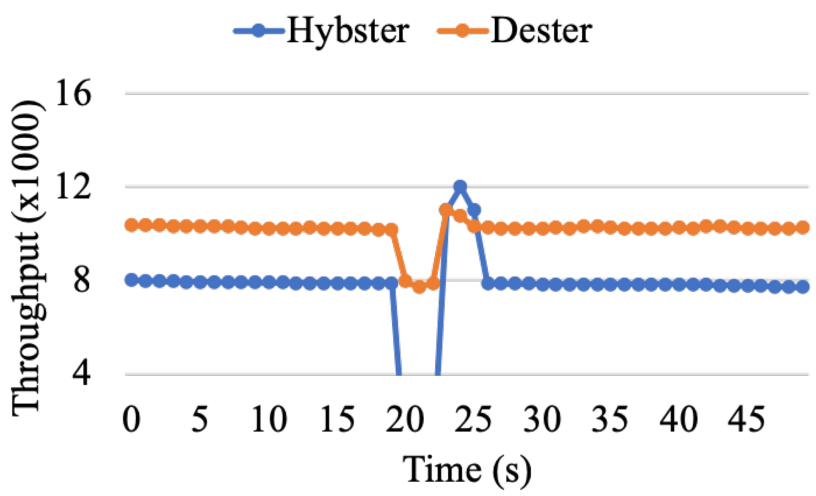


Figure 5.5: DESTER Fault Tolerance.

in throughput for few seconds for DESTER, but a total loss of throughput in Hybster for a longer duration.

Chapter 6

Bumblebee

Byzantine Fault-Tolerant (BFT) SMR solutions [12, 16, 27, 30, 33, 34, 38, 45, 52] can defend applications from arbitrary faults. However, such solutions are seldom used in practice due to their high costs – they require $3f + 1$ nodes to tolerate f arbitrary failures. For many replicated systems, such as those maintained by a single enterprise-class organization within a private network, malicious attacks, where the adversary can take control of faulty machines as well as the network in a coordinated way, are essentially non-existent [64]. This makes BFT protocols superfluous in such cases.

Prior work has proposed special consistency verification processes [68] and fault-recovery mechanisms [14] to detect or tolerate non-crash, non-malicious failures. However, this adds unnecessary complexity to layers outside of the core consensus layer. In recent works, a small trusted subsystem that can only fail by crashing has been shown to reduce the number of replicas required for *safe* replication from $3f + 1$ to $2f + 1$, matching the CFT requirement. Such protocols are said to adopt the hybrid fault tolerance model, and are therefore called Hybrid Fault-Tolerant protocols [26, 32, 51, 61, 85]. The advent of trusted execution environments (TEE) within commodity hardware such as Intel SGX [37] and ARM Trustzone [63] has significantly reduced the entry barrier for hybrid protocols. A software-based subsystem can be safe-guarded within these environments providing the required level of trust.

Hybrid protocols have been proposed either as a "ground-up" construction [26] or as a transformation of existing BFT protocols [32, 85]. Both approaches involve a complex design process that will likely hinder their practical adoption. In addition, BFT protocols employ complex message passing and verification mechanisms, and they are not as widely used as CFT protocols such as Raft or Paxos.

On the other hand, significant research investment has been made for the CFT model and a rich body of CFT protocols exist (e.g., [13, 20, 56, 57, 60, 65, 73, 77, 78]). CFT protocols have been well understood (e.g., Raft) as well as widely implemented (e.g., Paxos, Raft, Zab). This raises the question: is it possible to transform CFT protocols to tolerate arbitrary faults using a trusted component, via a general methodology? If that can be done, then it would pave the way for a large class of hybrid protocols that tolerate arbitrary faults to be easily rolled out, leveraging the rich body of CFT protocols. Many existing SMR systems (e.g. [4, 7, 24, 35, 49]) already implement some CFT protocol, thus hybridization is more likely, owing to the benefits of such a transformation.

This chapter affirmatively answer this question. We identified the fundamental design differences between CFT, BFT, and Hybrid (BFT/TEE) protocols, and develop key insights that are necessary for a CFT-to-Hybrid transformation. We develop a trusted subsystem, called, TruCount, that enables CFT-to-Hybrid transformations. Armed with the insights and TruCount, we develop a general methodology, called Bumblebee, for transforming CFT protocols into Hybrid protocols. We apply Bumblebee for transforming popular CFT protocols, Raft, Paxos, and WPaxos [13] into their hybrid counterparts.

To understand Bumblebee's overheads, we developed implementations. We realized the TruCount subsystem using Intel SGX SDK, and implemented the hybrid versions of Raft, Paxos, and WPaxos, and evaluated them on SGX-capable virtual machines in the Azure cloud infrastructure [2]. We also modified Raft's (CFT) implementation in etcd [5], a widely used key-value store, and implemented our transformed Hybrid Raft protocol. Our experimental evaluations reveal that the overheads due to hybridization in the context of a system such as etcd is less than 30% in terms of system throughput.

Contributions. The chapter's core research contribution is the Bumblebee methodology for CFT-to-BFT transformations. Our work shows that existing CFT protocols can be easily enhanced to provide greater fault tolerance by incorporating a trusted component and changing some parts of the protocol. In doing so, we show that such transformations are simple enough that they can be easily understood, which may enable their greater practical adoption. Additionally, this chapter make the following contributions:

- We describe, with examples, the fundamental design differences between BFT, CFT, and Hybrid classes of protocols.
- TruCount. We propose a subsystem realized using trusted execution environments that serves as the basis for our transformations, and is key in reducing the number of communication steps as well as the number of exchanged messages (§ 6.1.1).
- *Hybrid Protocols.* We show the feasibility of Bumblebee approach by transforming Raft, Paxos, and WPaxos to their hybrid versions. (§ 6.1.2).
- *Persistence and Integrity.* We show that our TruCount-dependent transformed solutions can intrinsically prevent rollback attacks [67] that are ubiquitous in trusted execution environments (§ 6.5).
- *Correctness.* We have formally specified and model checked the transformed hybrid protocols in TLA+ [55] to ensure that the protocols satisfy the safety properties including Consistency and Stability, and guarantee liveness. (§ 6.6).
- *Evaluation.* By implementing and experimentally evaluating the original and transformed protocols, we show that protocol transformations only require a nominal effort from well-seasoned developers. Furthermore, we claim that the achieved safety and security properties outweigh the performance penalties of hybrid protocols (§ 6.7).

6.1 BUMBLEBEE

We now present the BUMBLEBEE methodology for transforming CFT protocols into BFT protocols. We first discuss the concept of our trusted subsystem, and then proceed with a high-level overview of the transformation.

6.1.1 The Trusted Subsystem

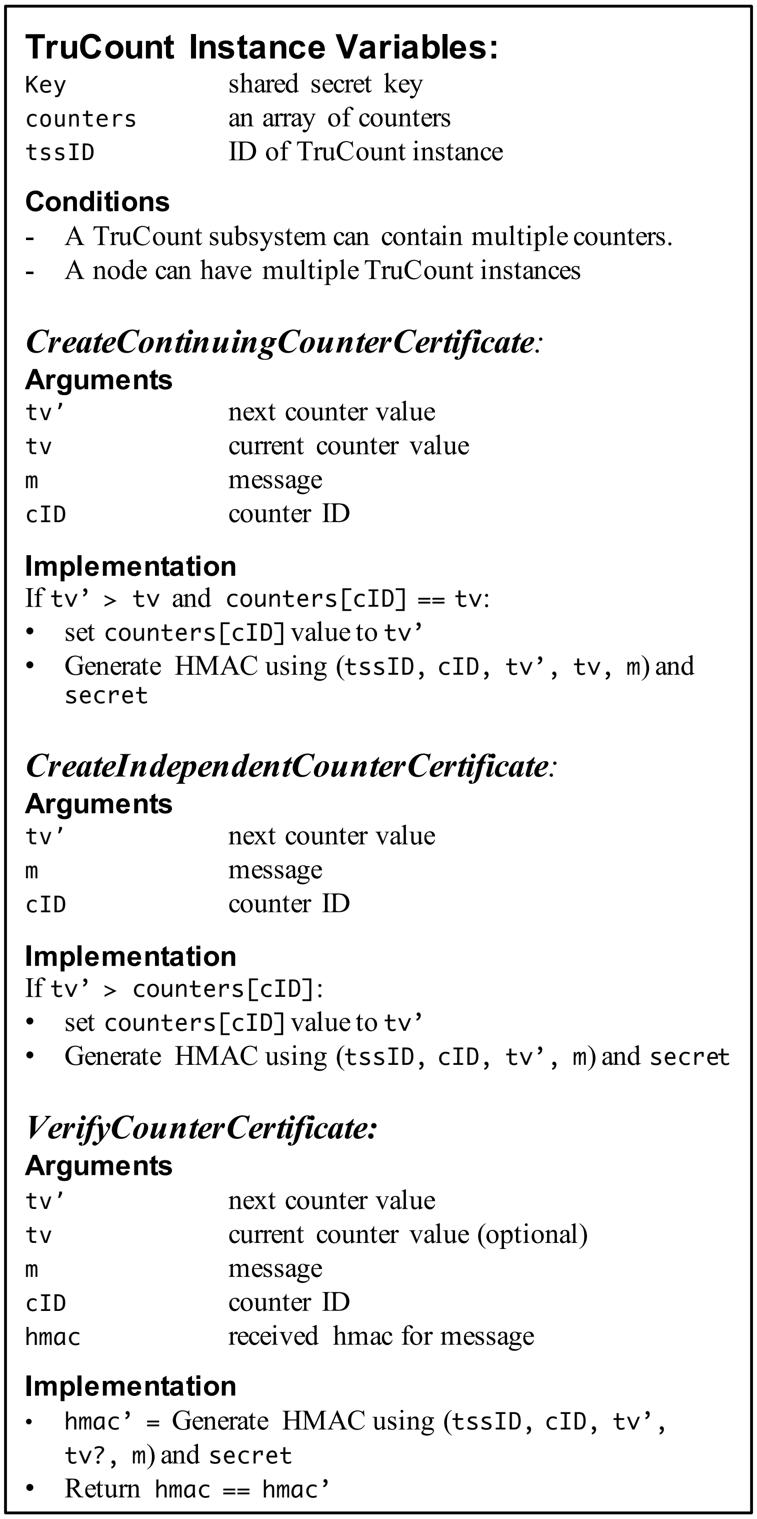
Recent commodity processors provide the so called *trusted execution environment* such as Intel SGX [37] and ARM Trustzone [63] that can run arbitrary software inside a protected enclave. Code running in an enclave is protected from undesired accesses by other software, even the operating system. This opens up new possibilities for creating trusted subsystems using software, unlike prior approaches that used dedicated ASICs [61] or FPGAs [51]. The wide availability of such trusted hardware capability at the commodity-scale increases the potential of the BUMBLEBEE methodology for practical adoption.

There exists multiple proposals for trusted subsystems in the literature [26, 32, 51, 61, 85]. One of the fundamental design goals for a trusted subsystem is simplicity, because simpler the component, the easier it is to verify its trust. Therefore, we propose TRUCOUNT, a counter-based subsystem that is inspired by TrInX [26] and USIG [85] subsystems.

TrInX takes an explicit counter value and a message, and certifies the message only if the counter value is greater than or equal to any previous observed value. To verify certificates, the recipient must regenerate certificates using the TrInX and verify it outside the subsystem. Furthermore, TrInX provides multiple interfaces specifically tailored for Hybster and are not generally applicable. Unlike TrInX, USIG implicitly increments the counter values and provides specific interfaces for creating and verifying certificates.

TRUCOUNT achieves the simplicity and expressiveness of USIG's interfaces, while exposing the ability to explicitly set counter values. The counter value represent crucial protocol variables (see Section 6.1.2), thus explicitly setting them simplifies the transformation process (such as the view change protocol). Furthermore, well-defined interfaces for certification and verification streamlines protocol development.

We assume, for the sake of simplicity, that all instances of TRUCOUNT are initialized appropriately by a trusted administrator before starting the state machine. Furthermore, each TRUCOUNT instance is assigned an identifier $tcID$, initialized with the required number of counters, and set up with a secret key that is shared among all TRUCOUNT instances. The counters are all set to 0. Moreover, we assume that the trusted execution environment (e.g., Intel SGX) is prone to undetected replay attacks where an adversary saves the state of a trusted subsystem and starts a new instance using the exact same state to reset the subsystem. We describe prevention techniques in Section 6.5. Figure 6.1 describes the functions provided by TRUCOUNT.



TruCount Instance Variables:

`Key`	shared secret key
`counters`	an array of counters
`tssID`	ID of TruCount instance

Conditions

- A TruCount subsystem can contain multiple counters.
- A node can have multiple TruCount instances

CreateContinuingCounterCertificate:

Arguments

`tv'`	next counter value
`tv`	current counter value
`m`	message
`cID`	counter ID

Implementation

If `tv’ > tv` and `counters[cID] == tv`:

- set `counters[cID]` value to `tv’`
- Generate HMAC using (`tssID, cID, tv’, tv, m`) and `secret`

CreateIndependentCounterCertificate:

Arguments

`tv'`	next counter value
`m`	message
`cID`	counter ID

Implementation

If `tv’ > counters[cID]`:

- set `counters[cID]` value to `tv’`
- Generate HMAC using (`tssID, cID, tv’, m`) and `secret`

VerifyCounterCertificate:

Arguments

`tv'`	next counter value
`tv`	current counter value (optional)
`m`	message
`cID`	counter ID
`hmac`	received hmac for message

Implementation

- `hmac’ =` Generate HMAC using (`tssID, cID, tv’, tv?, m`) and `secret`
- Return `hmac == hmac’`

Figure 6.1: Summary of the TRUCOUNT Subsystem.

- ***CreateContinuingCounterCertificate***. This function is used to create a certificate indicating the transition of a counter's value to a new higher value.
- ***CreateIndependentCounterCertificate***. This function is used to create a certificate with a new counter value that is higher than any previous value, but the actual past value is not important and is not encoded in the certificate.
- ***VerifyCounterCertificate***. This function is used to verify a certificate produced by another TRUCOUNT instance.

6.1.2 Overview of Transformation

Client-Replica Interaction

Existing CFT protocols must first be modified such that multiple replicas reply to a client's request. Clients must send request to a replica and connect to all replicas to receive a reply. However, this will require significant changes to the client-side implementation. This is where a proxy such as [62] can be helpful: the client can send a request to a proxy and wait for replies; the proxy layer can collect and validate the responses.

Replica-side implementation must be modified such that every replica replies to the client upon execution. This change is minimal, as every replica execute the client request and produce a result. Thus, the only change required is for replicas to forward the reply to an appropriate client.

Replica-side

The BUMBLEBEE approach to transformation begins with applying the optimizations for latency, i.e., reducing the communication steps, to existing CFT solutions (see Section 2.4). When replicas start communicating with each other, they can all collect quorums, execute request, and return the response back to the client at the same time as the leader.

The aforementioned changes are minimal enough that it does not change the core correctness arguments of the CFT protocols and therefore, they can still be proved correct in the hybrid model without significant additional effort.

The next step in the transformation is identifying the fundamental variables that track progress and serve as the basis upon which safety and progress conditions of the protocol depend upon. For Paxos, these variables are `Ballot` and `Instance Number`; for Raft, they are `Term` and `Index`; and so on. After identification, these variables are flattened and mapped to a counter in an instance of TRUCOUNT. `Term` and `Index` can be flattened by shifting `Term` towards the most significant bits and OR-ing with `Index`.

The changes to the agreement phase are straightforward: every replica-sent message is attested using the *Independent Counter Certificate* and is verified by the receiving replicas before operating on the message. The leader-election (or view change) transformation includes additional precautions necessary to ensure progress under the presence of byzantine replicas. Particularly, leader replicas are not allowed to participate in consecutive re-elections to ensure that byzantine leaders do not gain an unfair advantage and use that to cease progress. Furthermore, the transformation respects the constraints under which the original protocols elect their leaders. Specifically, in Raft, a candidate receives a vote only if it's logs are as up to date as the voter, whereas, in Paxos, the voter helps a candidate to become up to date by transferring logs during the voting process. These subtle but nuanced differences among original protocols are respected in their transformed counterparts along with the re-election constraint that we specifically developed for BFT protocols.

Recall from Section 2.4 that, traditionally, in BFT and Hybrid protocols, the term (or view) numbers determine the leader for that particular term rather than replicas competing for the leadership. In contrast, CFT protocols allow for a democratic election, where multiple replicas can participate and collect votes at any given term, and only the replica with the highest votes is elected as the leader. This process may yield an unfair advantage to byzantine replicas; they may be re-elected again and again, but they will fail to make progress (i.e. propose requests). To prevent this, we propose a constraint to prevent byzantine replicas from being re-elected consecutively while enabling democratic elections under the Hybrid model. Such a constraint is the following: an elected replica cannot participate in an election for at least $f+1$ consecutive terms following its last successful re-election. An evicting FIFO queue of size f can be used to maintain a blacklist of replicas. Furthermore, byzantine replicas can spuriously request votes for electing itself as the leader by moving to higher terms, even when a stable leader exists. To curb this, correct replicas only respond to vote requests if at least $f+1$ other replicas suspect a leader.

6.2 Raft Transformation

We now present Hybrid Raft, a transformed Raft protocol that uses the TruCount trusted component to tolerate byzantine faults. Hybrid Raft follows a similar message exchange pattern as that of Raft, but requires replicas to tolerate byzantine faults. The Hybrid Raft algorithm is composed of two principal components similar to its CFT counterpart: leader election (Section 6.2.2) and log replication (Section 6.2.1). Figure 6.2 provides and contrasts the steps taken by Hybrid Raft with respect to the original version.

Hybrid Raft provides the following properties:

1. *Election Safety.* At most one leader can be elected in a given term. An elected leader cannot face/win re-election for at least $f+1$ consecutive terms.
2. *Leader Append-only.* A correct leader never overwrites or deletes entries in its log; it only appends new entries.
3. *Log Matching.* If two logs contain an entry with the same index and term, then the

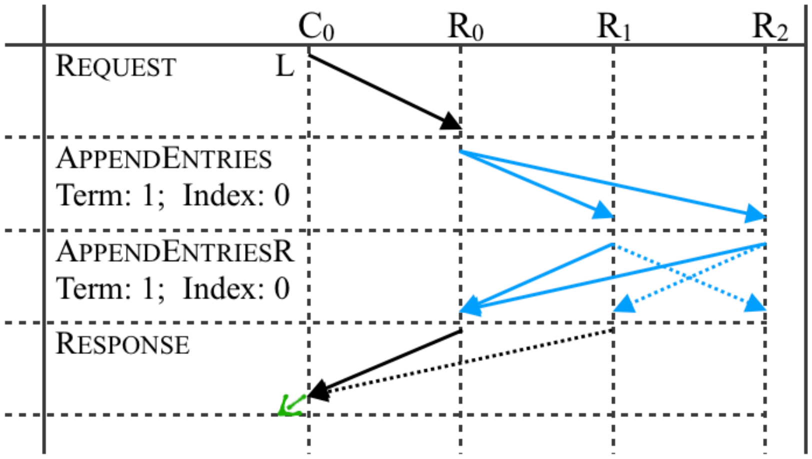


Figure 6.2: Raft vs Hybrid Raft: Normal case execution scenario. The dotted arrows show additional messages for Hybrid Raft. Notice `AppendEntriesR` and `Response` messages.

logs are identical in all entries up through the given index.

4. *Log Completeness.* If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
5. *State Machine Safety.* If a correct replica has applied a log entry at a given index to its state machine, then no other server will ever apply a different log entry for the same index.

6.2.1 Log Replication

A raft client c sends to the leader a signed $\langle \textsc{Request}, L, t, c\rangle_{\sigma_c}$ message containing the command L to be executed by the replicated state machine. The client includes a timestamp t to ensure exactly-once execution.

When a leader receives the request from the client, it appends the command to its log and issues an attested `Append-Entries` RPC in parallel to each of the other replicas. The `AppendEntries` RPC is attested by invoking the *CreateIndependentCertificate* interface of TRUCOUNT. The counter value is obtained by flattening the term number and the index number into $[term \cdot index]$. The dot $\cdot$ denotes append.

Upon receipt of an `AppendEntries` RPC, a follower checks that the attestation is valid and that the new entries do not skip log indexes by comparing the $prevLogIndex$ field in the message to the latest local log index. If the attestation is valid and the new log indexes are a consecutive extension of existing ones, the follower appends the entries to its log. The follower then broadcasts `AppendEntriesResponse`, containing the counter attestation, to all replicas.

When a replica receives at least $f+1$ `AppendEntries-Response` RPCs for a particular index and term with valid counter attestations, it proceeds to commit the log entries. The replica executes the log entry on the replicated state and returns a response back to the client in a $\langle \text{Reply}, result, t, r \rangle$ message, where r is the replica identifier.

The client waits for $f + 1$ replies with matching *result* and t fields, and returns to the application.

6.2.2 Leader Election

wA leader election is triggered when follower replicas suspect the leader to be faulty. Whenever the client timeouts waiting for $f + 1$ Reply messages, it sends the Request message to all the replicas. If a replica has already applied the corresponding log entry for that request, it simply sends a Reply message to the client. If no such request was proposed by the leader, then the replica forwards the request to the leader and starts a timer. If, within the timeout period, the follower replica receives an `AppendEntries` RPC with the pending request, the follower follows the protocol normally. Otherwise, when the timer expires, the follower casts its doubt on the leader by sending a `RequestTermChange` RPC to all replicas after incrementing its term number. This marks the beginning of leader election.

The purpose of the `RequestTermChange` is to amass support for changing the leader and ensure that leader change happen only when required and prevent byzantine replicas from triggering spurious leader elections.

A replica may enter the candidate state as soon as it receives $f + 1$ `RequestTermChange` RPCs, after which it is free to send `RequestVote` RPCs. Replicas that were elected leaders in any of the previous f terms are prohibited from entering the candidate state. Despite this restriction, a byzantine replica may send a `RequestVote` RPC, but correct replicas will disregard this message after checking against a blacklist.

The transformed `RequestVote` RPCs function similar to their original versions, but are attested with continuing counter certificates. In obtaining the certificate, the counter value is updated from $[term \cdot prevLogIndex]$ to $[term' \cdot 0]$, where $term' = term + 1$.

A follower responds to a `RequestVote` RPC if it has not responded to any such request for the same term and if the candidate logs are at least as up to date as its logs. Byzantine followers cannot misbehave (i.e. vote for two candidates in the same term) since their responses must be attested with a *Continuing Counter Certificate* after updating the TruCount's counter value from $[term \cdot prevLogIndex]$ to $[term' \cdot 0]$.

Once a leader has amassed a quorum of votes, it sends an `AppendEntries` RPC with a proof that it has received a quorum of votes in the `RequestVote` RPC. This `AppendEntries` RPC is special in that it establishes the sender as the leader and updates the TruCount's counter values in all the nodes from $[term' \cdot 0]$ to $[term' \cdot prevLogIndex]$. The followers add the leader to their re-election blacklist and enter the log replication state.

6.3 Paxos Transformation

We now describe the transformations to the Paxos protocol, specifically multidecree Paxos [82] that uses a designated leader, to produce *Hybrid Paxos*. *Hybrid Paxos* uses the same number of nodes and communication steps as Paxos, to reach agreement in the presence of a stable leader. Paxos's leader election is replaced with a hybrid mechanism that can withstand malicious replicas. The new protocol requires $2f+1$ replicas to tolerate f arbitrary failures, while guaranteeing safety and liveness. We begin with a description of the agreement component in the following subsection, and defer details of leader election to Section 6.3.2.

6.3.1 Agreement

We start from the optimized version of Paxos, where the explicit `Commit` message is replaced by broadcasting `AcceptAck` messages to all replicas in prior step (see Section 2.4), to carry out the transformation to *Hybrid Paxos*. The *instance number* defines the order for a client command with respect to other commands, while the *ballot* number indicates the number of leader elections as well as the latest state of the protocol. The counter values for TRUCOUNT is obtained by flattening the *ballot* and *instance number*s into a single number space of the form $[b \cdot i]$, where *ballot* b is stored in a fixed number of most significant bits and *instance number* i is stored in the rest of the least significant bits. The dot $\cdot$ denotes append.

The transformed protocol, *Hybrid Paxos*, works as follows. The leader sends a `Accept` message with the instance number i, ballot b, and the command c, along with a *independent counter* certificate from TRUCOUNT for $[b \cdot i]$. This ensures that the leader cannot create a lower or equal counter value than $[b \cdot i]$.

The replicas receive the `Accept` message and verifies the certificate of the message. The replicas then send a `AcceptAck` message to all replicas with its updated counter value $[b \cdot i]$, indicating an acknowledgment of the command at instance number i and ballot b. The `AcceptAck` is also certified using TRUCOUNT. The replicas, upon receipt of $2f+1$ `AcceptAck` messages, verifies the certificates, and executes the command and replies to the client. The client waits for $f+1$ equal responses and returns to the application.

Observe that this transformation yields a protocol equivalent to Hybster's agreement protocol [26].

6.3.2 Leader Election

The original leader election mechanism of Paxos must be modified to cope with byzantine replicas. The new mechanism adopted for Hybrid Paxos is very similar to that of Hybrid Raft, with a minor modification. Since, unlike Raft, Paxos allows any replica to be elected leader irrespective of the freshness of their state, we provide the same capability in Hybrid Paxos. This means that whenever acceptors support candidate proposers in an election, they must also send the missing logs in the candidate's state. Furthermore, an elected proposer must prove that it has the updated state by sending the received attested *view change* messages to other replicas. A complete description is provided in the technical report [9].

6.4 WPaxos Transformation

We now present Hybrid WPaxos, a hybrid fault-tolerant protocol obtained by transforming WPaxos [13] using TruCount. WPaxos is a multi-leader consensus protocol that uses an object-specific ownership mechanism to provide low latency and high throughput in wide area network (WAN) deployments. WPaxos extends M^2Paxos [78] by adopting the flexible quorum idea to reduce the number of replicas required for a quorum. The core novelties of WPaxos are: (i) an object-specific ownership mechanism wherein a replica can propose an order for commands if it owns the accessed objects; (ii) an instantiation of the flexible quorum, called grid quorum [47], where replicas are arranged in a grid form and the quorums are obtained as a combination of rows and columns. The advantage of such an approach is a smaller quorum size than majority quorums leading to faster commit times; and (iii) fast adaptation to changing access patterns, particularly common in geographically-wide deployments, by stealing object ownership from other replicas.

Hybrid-WPaxos provides the same novelties as WPaxos, but increases the strength of the fault-model from crash faults to hybrid faults (backed by TruCount).

6.4.1 Grid Quorums

Hybrid-WPaxos uses the same grid quorums that WPaxos employed. In a grid quorum system, the replicas are arranged into rows and columns forming a grid. The columns are called zones, which can represent a unit of availability or a geographical location. From the grid, two forms of quorums, namely Q_1 and Q_2 are extracted. The Q_1 quorum consists of $f+1$ replicas per zone of $2f+1$ replicas, over $Z-F$ zones, where Z is the number of zones, F is the number of zone failures, and f is the number of zonal failures. The Q_2 quorum consists of $f+1$ replicas per zone over $F+1$ zones. Figure 6.3 shows an example of grid quorums.

Due to the equivocation prevention capability of TruCount, the grid quorum can also guarantee safety in the hybrid fault model. The WPaxos quorums Q_1 and Q_2 intersect in at least one zone, and since a majority of the replicas are taken in the common zone, we have a non-empty intersection. As pointed out in [26, 85], even if the intersecting replica is byzantine, it cannot equivocate without being detected due to the continuing counter certificate that is attached during *ownership* (view) changes.

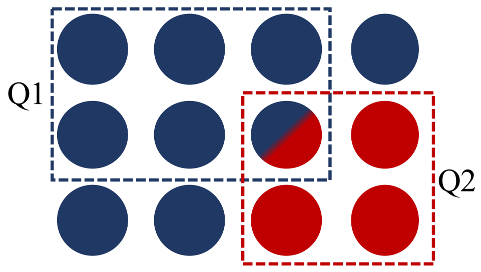


Figure 6.3: An example of WPaxos grid quorums that is also applicable for the hybrid variant.

6.4.2 Protocol

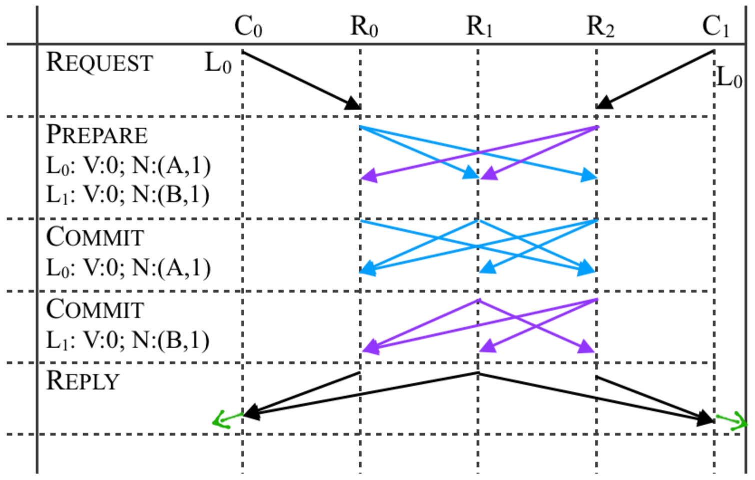


Figure 6.4: Hybrid WPaxos execution. R_0 and R_2 own objects A and B, respectively.

WPaxos uses an instance of the Flexible Paxos protocol [47] per object. Hybrid WPaxos uses an instance of the Hybrid Flexible Paxos protocol instead. This protocol can be obtained by adopting the grid quorums described earlier with each instance of Hybrid Paxos. Otherwise, the agreement protocol of Hybrid WPaxos largely remains the same.

An example execution is presented in Figure 6.4. In the example, two client send concurrent but non interfering requests to two different replicas. The replicas that receive the request (fortunately) have the ownership of the objects accessed by their requests. Thus, they independently propose their commands to commit them using their respective Hybrid Paxos instances. The two requests can be executed in any order because after execution of those requests in any order in any replica, the eventually state of the system will remain the same.

The ownership transfer in WPaxos involves executing the leader election sub-protocol and seeking acknowledgement from a quorum of replicas to gain ownership on a set of objects. WPaxos uses the same leader election protocol as Paxos, thus it is enough to use the same protocol as described in Section 6.3.2. However, we should relax the constraint that an *owner* (leader) cannot participate in re-elections for at least $f+1$ consecutive terms. Consider, for instance, the following scenario. Replica R_0 relinquishes its ownership to another replica R_1 temporarily, and wants to take ownership again. With the current protocol, this is not possible because R_0 cannot win an election for at least the $f+1$ terms. To address this situation, we introduce a pre-acquisition step where ownership-seeking replica R_0 persuades other replicas to remove it from their blacklists.

In Hybrid WPaxos, since at least $f+1$ replicas reply to the client, they know the client location, and can thus validate if a potential ownership transfer is required. A replica initiates the ownership change by sending its client access metadata to all replicas in a `Whitelist` message. Another replica, receiving this message, verifies the received metadata using its local metadata. If the metadata match, then the replica sends its `Whitelist` message. Once replicas have received at least $f+1$ such identical messages, they remove the concerned replica from the re-election blacklist. Following, ownership acquition proceeds as in Hybrid Paxos. A complete description is provided in the technical report [9].

6.5 Persistence and Integrity

Consensus protocols persist some information to stable storage to cope with failures. Consequently, our transformed hybrid protocols must also persist critical state information including the TruCount's state to stable storage, thus making them susceptible to rollback attacks [67]. A byzantine replica can restart the TruCount instance and provide it with a stale snapshot of the counter state from the stable storage, thus successfully executing a rollback and violating the integrity of the enclave's state. Since the trusted execution environment cannot distinguish between different versions of state independently, it has been shown in [67] that a distributed system can be used to efficiently prevent such attacks. The proposed solution seals the enclave state along with a monotonically increasing in-memory counter and uses a version of consistent broadcast protocols to distribute the counter value on a set of replicas. During retrieval, after server/enclave restarts, the counter value is summoned from other nodes and the integrity of the sealed data is verified.

For our specific case of consensus protocols where the TruCount's counter value is already replicated among a set of nodes, we propose a much simpler technique to store and retrieve the persistent state. The protocol messages (e.g. `AppendEntries` RPC in Raft) that are attested by TruCount encapsulates all the information that require persistence. Thus, it is enough to store the data along with the certificate before sending the messages to other nodes.

Whenever the consensus protocol is restarted (e.g. after a machine restart), TruCount reconstructs its state by obtaining its counter value from $f+1$ nodes and picking the highest value. Due to asynchronous networks and use of quorum, not every node will have the latest counter value, but one of $f+1$ is guaranteed to have, since $N = 2f+1$ and any two quorums will have a non-empty intersection.

6.6 Correctness

We have formally specified the transformed protocols in TLA+ [55]. Using the TLC model checker, we verified that the specifications hold their safety and liveness properties. We specified the characteristics of byzantine replicas following the example in [58]. Furthermore, we specified the TRUCOUNT subsystem under the assumption that it is safe from byzantine behaviors; that is, we excluded byzantine replicas from tainting the subsystem. The complete TLA+ specifications of the transformed protocols and their invariants are available in a technical report [9].

6.7 Evaluation

We implemented the proposed hybrid solutions, Hybrid Paxos, Hybrid Raft, and Hybrid WPaxos (and also Hybster) in Go. The trusted subsystem TRUCOUNT was implemented in C using Intel SGX SDK and was interfaced with the Go application using `cgo` [43]. All protocols, with the exception of Hybrid Raft, was implemented in a common framework. We implemented Hybrid Raft by applying the transformations to etcd's Raft implementation [5].

A note on Hybrid Raft. In etcd's Raft, a leader may send `AppendEntriesRPC` containing different number of entries with varying index ranges to different replicas. That is, a leader may send entries indexed 1 through 5 to one replica, and send entries 3 through 5 to another replica. Because of this inconsistency, creating certificates for the counter values is a challenge. To address this, we cache the certificate and their counter values. If a certificate has already been created for part of the index range, then the leader splits and sends two `AppendEntriesRPC`s, one with the cached certificate and another with the new certificate.

We deployed all the systems on a cluster of three virtual machines in the Azure infrastructure [2]. Each VM belong to the DC2-series and has a 3.7GHz Intel Xeon E-2176G Processor with 8GB RAM. The OS used was Ubuntu 16.04 with Intel SGX SDK v2.3.1 and Go v1.12 installed.

6.7.1 Implementation Complexity

To validate our claim that our approach is more practical than any ”ground-up” construction of consensus protocols, we analyzed the effort required to transform each of our candidate original protocols to their transformed versions. The prototype versions of Paxos and WPaxos were the easiest to transform as their source code were only about 500 lines and 800 lines, respectively. The transformed versions, Hybrid Paxos and Hybrid WPaxos, added/changed about 500-600 lines in each of the implementations. The SGX-based TruCount implementation was about 268 lines (excluding the auto-generated code). The transformation of Raft required more effort as it was a non-monolithic, high-performance design implementation. The whole transformation (including TruCount) added/changed about 1400 lines of code.

6.7.2 TruCount

First, we evaluated the performance of the TruCount subsystem in order to understand the overhead of SGX calls and cgo interfaces. We measured the throughput of creating the *independent counter certificates* for different message sizes under three different setups: (i) calling the SGX function from a C application, (ii) calling the SGX function from a Go application using the cgo interface, and (iii) generating independent counter certificates using untrusted Go code.

Figure 6.5 shows the results. The x-axis shows the message sizes and the y-axis shows the throughput in terms of megabytes of messages certified per second. We can observe that, for smaller messages (under 8KB), creating the Bumblebee certificates are much costlier than untrusted certification. In some cases, the difference is as much as 2.5x (for 1KB). However, the difference zeros out for 16KB messages and beyond. Also, note that the cgo interface adds negligible overheads.

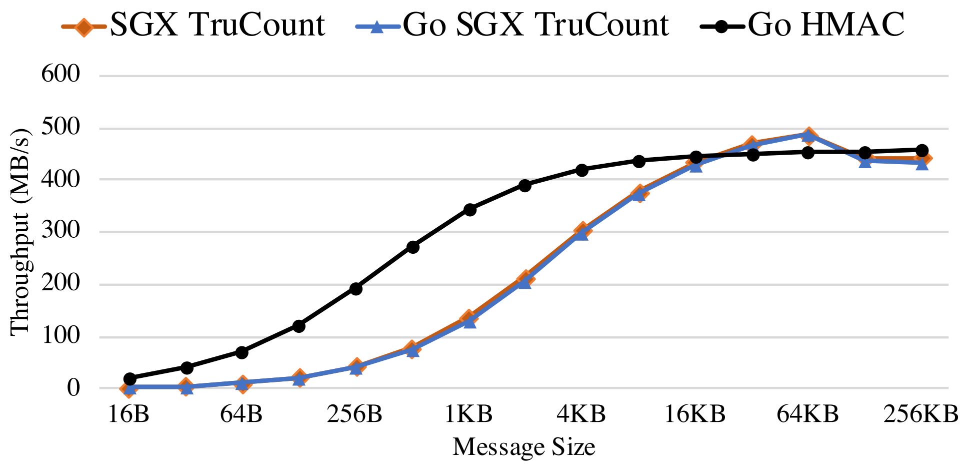


Figure 6.5: Throughput of SGX certifier vs Go-based certifier.

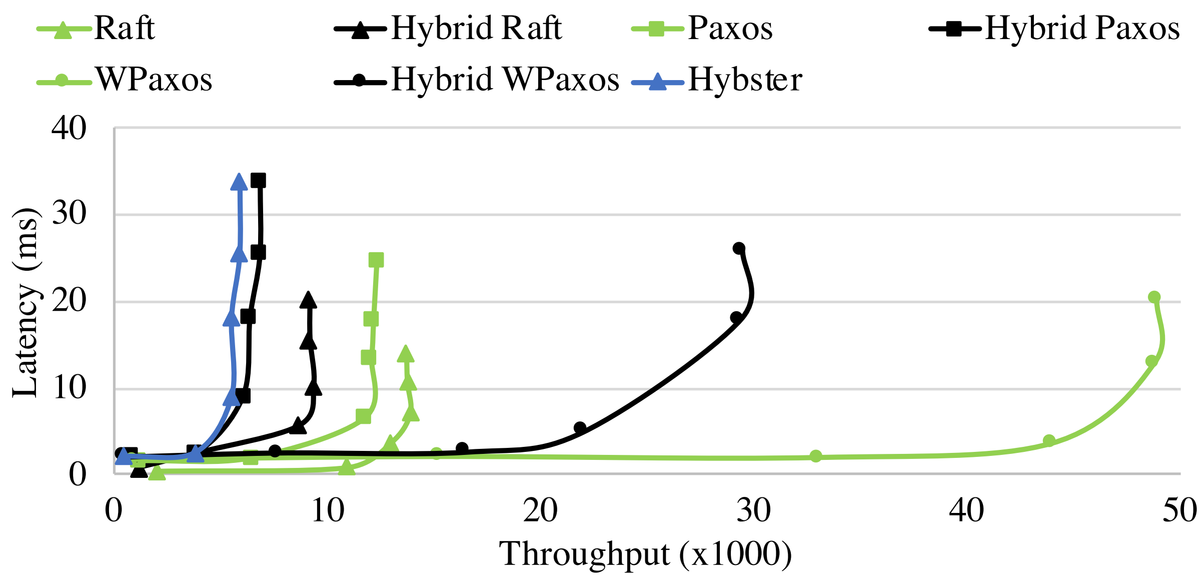


Figure 6.6: Throughput vs latency of Hybrid protocols.

6.7.3 Performance of Hybrids

Figure 6.6 shows the throughput versus latency of the original and the transformed variants of the CFT protocols. For this experiment, we varied the number of clients between 1 and 250, and injected requests into the system. We note that Hybster and Hybrid Paxos perform very similar, as they are equivalent protocols. On the other hand, Hybrid WPaxos has the advantage of using per-object Hybrid Paxos. Thus, non-conflicting commands can now be executed in parallel, thereby improving the overall throughput. In our experiment, the ownership acquisition was turned off, so Hybrid WPaxos replicas forward commands to the appropriate leader. Also, note that in a three replica system, the WPaxos grid quorum degenerates to a majority quorum. It can be observed that the hybrid variants incur a latency penalty in the range of 30% (for Hybrid WPaxos) to 45% (for Hybrid Paxos). This is due to the increase in the number of messages exchanged as well as the usage of TRUCOUNT to create attestations. Moreover, Raft and its hybrid variant has the best performance among single-leader protocols despite similarity. We attribute this to etcd's original high-performance implementation, which Hybrid Raft also exploits resulting in a performance penalty of only about 30%. Specifically, unlike Paxos that replicates a request to a quorum of nodes, Raft replicates requests on a per node basis and waits until a quorum has acknowledged. This enables replicas to be independently replicated and thus improve overall performance. Finally, we believe such penalties are acceptable given the security and integrity benefits of the transformed protocols.

6.7.4 Scalability of Hybrids

Figure 6.7 compares the scalability of Hybrid Raft and original Raft as the number of replicas increase. We repeated the throughput versus latency experiment (in Section 6.7.3) for 3, 5, 7, and 9-node setups. It can be observed that Hybrid Raft's performance penalty is about 30% in the 3-node setup and grows to about 50% in the 9-node setup. This can be attributed to the exponential increase (n^2) in the number of messages exchanged as well as the overhead of interfacing with the SGX enclave and creating attestations. However, in practice, this overhead can be minimized by batching protocol messages sent over the network as well as interfacing with the SGX for a batch of messages (due to low SGX overheads for large messages as shown in Figure 6.5).

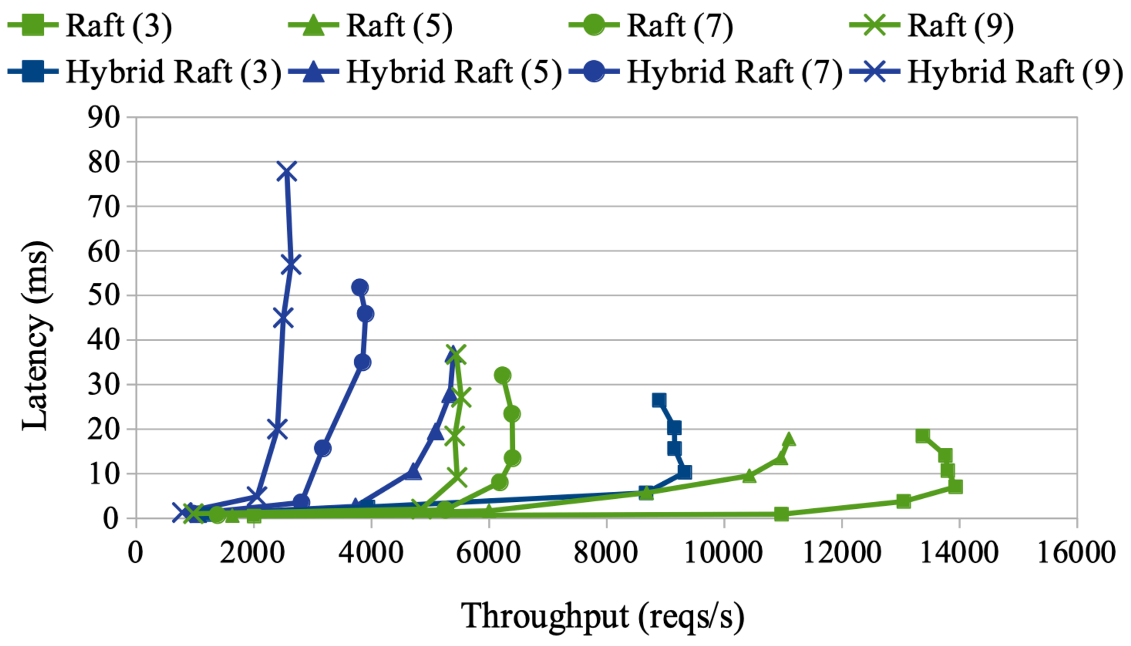


Figure 6.7: Throughput vs latency of Hybrid Raft vs Original Raft as number of nodes is varied. In the legend, the number inside the parenthesis indicate the number of nodes in that experiment.

6.7.5 Fault Tolerance of Hybrids

Our hybrid protocols behave similar to their CFT counterparts under normal conditions (non-faulty nodes and network). As soon as a non-leader is faulty, the throughput decreases by about 5–6% in a 3-node setup and by about one percent in a 7-node setup. In case of 3 nodes, the loss of a node increases the load on the remaining two nodes since they must bear all the responsibilities including being part of the quorum and responding to the clients.

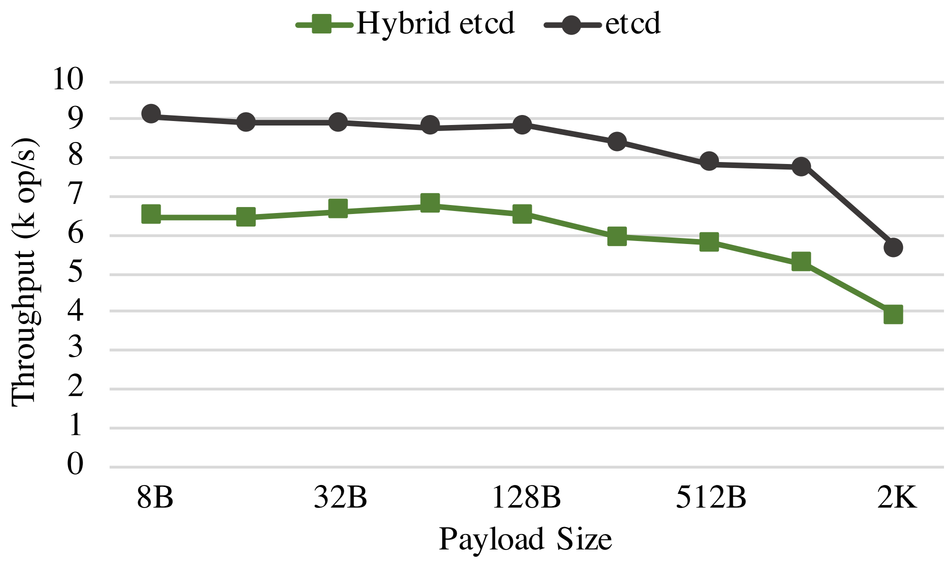


Figure 6.8: Original etcd vs Hybrid Raft etcd.

However, at 7 nodes, the loss of a node still leaves three additional nodes that can share the workload.

6.7.6 etcd

etcd [5] is a distributed key-value store that is widely used in numerous production systems [6]. It uses the Raft consensus protocol to manage its replicated log. We transformed etcd's Raft implementation and benchmarked it with etcd's benchmarking tool. We modified the tool to collect $f + 1$ replies for each request and added the capability for all replicas to reply to the client. We compared our transformed etcd against the original version.

For this experiment, we deployed three replicas of etcd, and instantiated 500 clients to send requests to the replicas. We varied the payload size and measured the throughput by sending 100K requests. Figure 6.8 shows the results. It can be observed that the maximum performance penalty of the hybrid approach is between 20–30%. etcd performs several optimizations including batching of entries, which creates large enough message sizes that nullify the impact of SGX calls (recall Section 6.7.2). This shows that the proposed Bumblebee approach is viable in today's systems, causing minimal performance degradation, while tolerating non-crash faults including hardware and software failures.

Chapter 7

Conclusions

With the ever increasing proliferation of distributed systems in practice, the need for consensus-based SMR solutions will only deepen. Crash fault tolerance has been the go-to fault model for building SMR-based systems due to stronger assumptions that make the underlying designs simpler. However, evidence from deployed systems has shown that the assumptions of the CFT model do not reflect the faults that occur in practice [36]. The proponents of BFT protocols have long made the case against CFT solutions [52], but practitioners were discouraged because consensus protocols are hard to get right; even more cumbersome is adapting a totally new consensus scheme in a system that was built around the original protocol [27]. Therefore, practitioners have usually built *adhoc* techniques to tolerate non-crash faults as they occur [23, 36, 68]. This approach creates unnecessary software bloat, increasing development, testing, and maintenance costs.

The BUMBLEBEE approach provides a methodology to enhance existing CFT implementations to tolerate arbitrary faults. The proposed transformation is not *adhoc*. Our intent is to enable system builders to transform their *in-house* consensus implementation to tolerate arbitrary faults, by following our general methodology. In doing so, we also address a critical trade-off: we propose to transform to the hybrid fault model instead of the byzantine fault model. This is because, transformation to the BFT model is not straightforward as there exists fundamental differences in the minimum number of replication nodes and communication steps necessary to guarantee safety and liveness properties for the respective models. On the other hand, hybrids have more in common with the CFT protocols, particularly in the number of communication steps and the degree of replication. Furthermore, the trusted environment required to implement TRUCOUNT are available both in cloud systems [2] and in commodity processors [37], paving an easier path for practical adoption.

Furthermore, this dissertation proposed two leaderless consensus protocols, one each for the Byzantine and Hybrid fault models. Existing hybrid protocols provide poor performance and availability guarantees in geo-scale deployments, even though CPU-based Trusted Execution Environments have made state machine replication solutions based on hybrid fault model more appealing than ever. Therefore, we present DESTER, a leaderless hybrid fault tolerant state machine replication protocol that provides optimal latencies for clients located at different geographical regions.

State-of-the-art BFT protocols are not able to provide optimal request processing latencies in geo-scale deployments – an increasingly ubiquitous scenario for many distributed applications, particularly blockchain-based applications. We presented EZBFT, a leaderless BFT protocol that provides three-step consensus in the common case, while essentially nullifying the latency of the first communication step. EZBFT provides the classic properties of BFT protocols including nontriviality, consistency, stability, and liveness.

7.1 Proposed Work Post Preliminary Examination

The following subsections presents avenues of future work post the preliminary examination. Section 7.1.1 propose to investigate automatic code and proof generation in order to increase the feasibility and approachability of the BUMBLEBEE approach. Section 7.1.2 proposes to develop highly scalable byzantine fault tolerant asynchronous state machine replication protocols for cryptographic currency usecases.

7.1.1 Automatic Code and Proof Generation

Posterior to the preliminary examination, this dissertation proposes to investigate methods to automate the BUMBLEBEE transformations. Currently, the BUMBLEBEE approach is largely a mechanical technique. Thus, it may be possible to auto-generate the transformations. Given a specification of a CFT protocol in a formal language (e.g. TLA+ [55], Verdi [88]), an algorithm should recognize the key pieces of the protocol and produce a specification of the hybrid fault-tolerant version. Given the complexities implicit in many consensus protocols, such an approach would minimize the time required for such transformations and minimize unnecessary bugs that could be induced with manual methods.

We propose an algorithm with three key functions. First, it should identify whether the input CFT algorithm is compatible with the BUMBLEBEE approach. This is because our technique does not apply to all existing CFT protocols particularly multi-leader variants (e.g. EPaxos). Second, the algorithm must recognize the key state variables of the candidate protocol that would serve as the basis for the counter TRUCOUNT. Currently we depend on an expert to identify these variables since these variables are difficult to recognize. Third, the algorithm must identify the agreement and leader election faces clearly and apply the transformations as described in Chapter 6.

Furthermore, it is important to establish the correctness of the transformed protocols and verify that they indeed provide the necessary safety and liveness properties. We propose to investigate methods to automatically generate machine checked proofs from the transformed specifications. Currently, tools such as TLAPS [69, 71] and Verdi exist that enable developers to write machine-checked proofs. Tools such as Isabelle/HOL have the capability to produce proofs for invariants from formal specifications [28]. This dissertation proposes to explore the potential of these tools to produce proofs for our specifications of transformed protocols.

More importantly, consensus protocols are hard to get right – both in terms of specifications as well as implementations. Therefore, this dissertation proposes to investigate techniques to automatically generate highly performant low level implementations from correct high-level specifications. Existing tools such as Verdi is capable of producing implementations in the OCaml [75] programming language. On the other hand, tools such as PGo [40] produce Go implementations from high level TLA+ specifications. Our goal is to produce understandable and debuggable implementations in widely used distributed systems programming languages such as Go. PGo is not capable of generating distributed system implementations, but multithreaded programs only.

7.1.2 Highly Scalable Asynchronous Byzantine Fault Tolerant Protocol

Most byzantine fault tolerant consensus protocols including those proposed in this dissertation are deterministic, meaning that these protocols can confirm the execution for a client request after executing a set of pre-determined steps. One of the major disadvantage of such protocols is that they provide very poor scalability deeming them unfit for cryptographic currency applications that typically require scalability at 1000s of nodes. Deterministic BFT protocols communicate with two thirds of the nodes in the system and exchange messages with each other. At scale, the number of messages exchanged for each request increases exponentially (see Section 6.7.4) essential becoming the bottleneck.

Cryptocurrency applications [74, 89] rely on Proof-of-Work based consensus solutions where participants compute cryptographic hashes to append entries to the blockchain (i.e. log). Such techniques are susceptible to *fork*ing problem, where different users have different views of the blockchain. To mitigate forks, a transaction must wait for a reasonably long time (e.g. upto one hour for Bitcoin) before commitment to ensure that the possibility of forks are negligible. This has resulted in poor transaction processing times.

Algorand [42], instead, proposes a byzantine fault tolerant consensus protocol that is highly scalable and can withstand malicious actors (thus preventing forks). Algorand achieves its goals by adopting a probabilistic consensus protocol and relying on an overwhelming majority (two thirds) of stakeholders to be honest. The protocol guarantees safety under the weak synchrony timing assumption. Under this assumption, the network can be asynchronous (i.e., entirely controlled by the adversary) for a long but bounded period of time (e.g., at most 1 day or 1 week), but after an asynchrony period, the network must be strongly synchronous for a reasonably long period of time (e.g., a few hours or a day) for Algorand to ensure safety.

However, in practice, the weak synchrony assumption is too strong [29]. Thus, many practical BFT protocols including PBFT and the contributions of this dissertation adopt the eventual synchrony timing assumption. Under the eventual synchrony assumption, the network is expected to be synchronous eventually, but the exact time is not known. Simply put, the assumption captures the fact that the system may not always be synchronous, and there is no bound on the period during which it is asynchronous. However, similar to the weak synchrony assumption, after the asynchrony period, the network is assumed to be strongly synchronous for a sufficiently long time to allow the protocol to make progress.

Therefore, we propose a byzantine fault tolerant consensus protocol under the eventually synchronous timing assumption. The principal challenge is achieving consensus with a high probability, similar to Algorand, but despite weaker timing assumptions. Algorand relies on the BA★ algorithm [70] to achieve consensus on transaction blocks. We propose to develop a probabilistic consensus algorithm, similar to BA★, that works under our new timing assumption.

Furthermore, Algorand proposes other techniques such as Weighed users and Consensus by committee to achieve its scale. These technique, however, are independent of the timing assumptions, thus they can easily be adopted for our proposed solution.

Bibliography

[1] Db instance replication. URL http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Overview.Replication.html. Accessed: 2017-01-25.

[2] Azure. https://azure.microsoft.com/, . Accessed: 2019-01-06.

[3] CosmosDB - Azure. https://azure.microsoft.com/en-us/services/cosmos-db/, . Accessed: 2018-11-06.

[4] CockroachDB. https://github.com/cockroachdb/cockroach. Accessed: 2018-11-06.

[5] etcd - CoreOS. https://coreos.com/etcd/, . Accessed: 2018-11-06.

[6] etcd - Production Users. https://github.com/etcd-io/etcd/blob/master/Documentation/production-users.md, . Accessed: 2019-01-05.

[7] Cloud Spanner - Google Cloud. https://cloud.google.com/spanner/. Accessed: 2018-11-06.

[8] gRPC. https://github.com/grpc/.

[9] Bumblebee Technical Report. https://www.dropbox.com/s/z4cxepxc33376y8/Bumblebee.pdf?dl=0.

[10] Overview: Sql database active geo-replication, September 2016. URL https://docs.microsoft.com/en-us/azure/sql-database/sql-database-geo-replication-overview.

[11] Daniel Abadi. Newsql database systems are failing to guarantee consistency, and I blame Spanner. http://dbmsmusings.blogspot.com/2018/09/newsql-database-systems-are-failing-to.html. Accessed: 2019-01-06.

[12] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095817. URL http://doi.acm.org/10.1145/1095810.1095817.

[13] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Multileader WAN paxos: Ruling the archipelago with fast consensus. *CoRR*, abs/1703.08905, 2017. URL http://arxiv.org/abs/1703.08905.

[14] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32. USENIX Association, 2018. ISBN 978-1-931971-42-3.

[15] Amazon Inc. Elastic Compute Cloud. http://aws.amazon.com/ec2.

[16] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, volume 00, pages 197–206, June 2008. doi: 10.1109/DSN.2008.4630088. URL doi.ieeecomputersociety.org/10.1109/DSN.2008.4630088.

[17] B. Arun and B. Ravindran. Achieving decentralization in hybrid fault tolerance. In *ACM Symposium on Cloud Computing 2019*, July 2019 In Progress.

[18] B. Arun, F. Verbeek, and B. Ravindran. Better fault tolerance for practical distributed systems. In *2019 The 27th ACM Symposium on Operating Systems Principles (SOSP)*, July 2018 Under Review.

[19] B. Arun, S. Peluso, and B. Ravindran. ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, July 2019.

[20] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017. doi: 10.1109/DSN.2017.35.

[21] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication, April 2019. URL http://arxiv.org/abs/1904.06023.

[22] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. 32(4):12:1–12:45, 2015. ISSN 0734-2071. doi: 10.1145/2658994.

[23] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20:20–20:32, July 2014. ISSN 1542-7730. doi: 10.1145/2639988.2655736. URL http://doi.acm.org/10.1145/2639988.2655736.

[24] J. Baker, C. Bond, J. Corbett, JJ Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2011.

[25] Rida Bazzi and Maurice Herlihy. Clairvoyant state machine replications. In Taisuke Izumi and Petr Kuznetsov, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 254–268. Springer International Publishing. ISBN 978-3-030-03232-6.

[26] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 222–237. ACM, 2017. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064213.

[27] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-2233-8. doi: 10.1109/DSN.2014.43. URL https://doi.org/10.1109/DSN.2014.43.

[28] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in isabelle/hol. In *Proceedings of the 8th International Conference on Frontiers of Combining Systems*, FroCoS'11, pages 12–27, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24363-9. URL http://dl.acm.org/citation.cfm?id=2050784.2050787.

[29] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 3642152597, 9783642152597.

[30] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://dl.acm.org/citation.cfm?id=296806.296824.

[31] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002. ISSN 0734-2071. doi: 10.1145/571637.571640. URL http://doi.acm.org/10.1145/571637.571640.

[32] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 189–204. ACM, 2007. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294280.

[33] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629602.

[34] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558988.

[35] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. 31(3):8, 2013.

[36] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342862.

[37] Victor Costan and Srinivas Devadas. Intel SGX explained. 2016(086):1–118, 2016.

[38] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190. USENIX Association, 2006. ISBN 1-931971-47-1.

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. 41(6):205–220. ISSN 0163-5980. doi: 10.1145/1323293.1294281.

[40] Minh Nhat Do. *Corresponding formal specifications with distributed systems*. PhD thesis, University of British Columbia, 2019. URL https://open.library.ubc.ca/collections/ubctheses/24/items/1.0378335.

[41] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL http://dl.acm.org/citation.cfm?id=646334.687813.

[42] 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*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132757. URL http://doi.acm.org/10.1145/3132747.3132757.

[43] Google. cgo - the Go programming language. http://golang.org/cgo, . Accessed: 2019-01-10.

[44] Google. Protocol buffers. https://developers.google.com/protocol-buffers/, .

[45] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 363–376. ACM, 2010. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755950.

[46] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1. URL http://dl.acm.org/citation.cfm?id=1387589.1387602.

[47] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, pages 25:1–25:14, 2016. doi: 10.4230/LIPIcs.OPODIS.2016.25. URL https://doi.org/10.4230/LIPIcs.OPODIS.2016.25.

[48] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11. USENIX Association.

[49] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[50] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, 1(1):36–63, August 2001. ISSN 1615-5262. doi: 10.1007/s102070100002. URL http://dx.doi.org/10.1007/s102070100002.

[51] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308. ACM, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168866.

[52] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. pages 45–58, 2007. doi: 10.1145/1294261.1294267. URL http://doi.acm.org/10.1145/1294261.1294267.

[53] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. 27(4):7:1–7:39, 2010. ISSN 0734-2071. doi: 10.1145/1658357.1658358.

[54] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication.

[55] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., . ISBN 032114306X.

[56] Leslie Lamport. Generalized consensus and Paxos, . http://research.microsoft.com/apps/pubs/default.aspx?id=64631.

[57] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006. URL https://www.microsoft.com/en-us/research/publication/fast-paxos/.

[58] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 211–224. Springer-Verlag, 2011. ISBN 978-3-642-24099-7.

[59] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. 4(3):382–401, 1982. ISSN 0164-0925. doi: 10.1145/357172.357176.

[60] Leslie Lamport et al. Paxos made simple. Technical report, 2001.

[61] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 1–14. USENIX Association, 2009.

[62] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rudiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 59–70, 2018. doi: 10.1109/DSN.2018.00019.

[63] ARM Limited. ARM security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. ARM Technical White Paper, Accessed: 2018-11-06.

[64] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500. USENIX Association, 2016. ISBN 978-1-931971-33-1.

[65] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384. USENIX Association, 2008.

[66] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. 3(3):202–215. ISSN 1545-5971. doi: 10.1109/TDSC.2006.35.

[67] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic.

[68] Vivek Menezes. Trust, but verify: How CockroachDB checks replication. https://www.cockroachlabs.com/blog/trust-but-verify-cockroachdb-checks-replication/, note = Accessed: 2018-11-06.

[69] Stephan Merz and Hernán Vanzetto. Harnessing smt solvers for tla+ proofs. In *12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*, volume 53. EASST, 2012.

[70] Silvio Micali. Fast and furious byzantine agreement. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2017.

[71] Microsoft Research–Inria. TLA+ Proof System. https://tla.msr-inria.inria.fr/tlaps/content/Home.html. Accessed: 2019-05-01.

[72] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2517350.

[73] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372. ACM, 2013. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2517350.

[74] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[75] OCaml Authors. OCaml Programming Language. https://ocaml.org/. Accessed: 2019-05-01.

[76] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM. ISBN 0-89791-277-2. doi: 10.1145/62546.62549. URL http://doi.acm.org/10.1145/62546.62549.

[77] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319. USENIX Association, 2014. ISBN 978-1-931971-10-2.

[78] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167, 2016. doi: 10.1109/DSN.2016.23.

[79] Fred B. Schneider. The state machine approach: A tutorial.

[80] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. doi: 10.1145/98163.98167. URL http://doi.acm.org/10.1145/98163.98167.

[81] Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri, and Binoy Ravindran. Be general and don't give up consistency in geo-replicated transactional systems. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 33–48. Springer International Publishing. ISBN 978-3-319-14471-9. doi: 10.1007/978-3-319-14472-6_3.

[82] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. 47(3): 42:1–42:36. ISSN 0360-0300. doi: 10.1145/2673577.

[83] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient byzantine agreement for wide-area networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 10–19, Nov 2010. doi: 10.1109/HASE.2010.19.

[84] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 135–144. IEEE Computer Society. ISBN 978-0-7695-3826-6. doi: 10.1109/SRDS.2009.36.

[85] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 62(1):16–30, January 2013. ISSN 0018-9340. doi: 10.1109/TC.2011.221. URL http://dx.doi.org/10.1109/TC.2011.221.

[86] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In Jan Camenisch and Doğan Kesdoğan, editors, *Open Problems in Network Security*, pages 112–125, Cham, 2016. Springer International Publishing. ISBN 978-3-319-39028-4.

[87] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 206–215, 2000. doi: 10.1109/RELDI.2000.885408.

[88] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737958. URL http://doi.acm.org/10.1145/2737924.2737958.

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