SCALING THE INFRASTRUCTURE OF PRACTICAL BLOCKCHAIN SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Maofan Yin

August 2021

SCALING THE INFRASTRUCTURE OF
PRACTICAL BLOCKCHAIN SYSTEMS

Maofan Yin, Ph.D.
Cornell University 2021

The infrastructure of a blockchain system consists of a replication service that tolerates limited adversarial behavior among participants. It requires both a Byzantine fault tolerant (BFT) replication protocol to defend against the adversaries and an underlying storage system to preserve states. This dissertation explores two designs for BFT replication and one design for the persistent storage.

We first present *HotStuff*, a leader-based BFT replication protocol for the partially synchronous system model. Once network communication becomes synchronous, HotStuff enables a correct leader to drive the protocol to consensus at the pace of actual (vs. maximum) network delay—a property called *responsiveness*—and with communication complexity that is linear in the number of replicas. To the best of our knowledge, HotStuff is the first partially synchronous BFT replication protocol exhibiting these combined properties. HotStuff is built around a novel framework that forms a bridge between classical BFT foundations and blockchains. It allows the expression of other known protocols (DLS, PBFT, Tendermint, Casper), and ours, in a common framework.

Our deployment of HotStuff over a network with over 100 replicas achieves throughput and latency comparable to that of BFT-SMaRt, while enjoying a linear communication footprint during leader failover (vs. cubic with BFT-SMaRt).

Then, we introduce a family of leaderless BFT protocols, exploiting metastable properties of network subsampling. These protocols provide a strong probabilistic safety guarantee in the presence of Byzantine adversaries while their concurrent and leaderless nature enables them to achieve high throughput and scalability. Unlike blockchains that rely on Proof-of-Work, blockchains built on our protocols are quiescent and green. Unlike traditional consensus protocols where typically one or more nodes must process a linear number of bits in the number of total nodes per decision, no node processes more than a logarithmic number of bits. It does not require accurate knowledge of all participants and exposes new possible tradeoffs and improvements in safety and liveness for building consensus protocols.

We describe the *Snow* protocol family, and how it can be used to construct the core of an internet-scale electronic payment system, *Avalanche*, which is evaluated in a large scale deployment. Experiments demonstrate that the system can achieve high throughput, provide low confirmation latency and scale well compared to existing systems that deliver similar functionality. For our implementation and setup, the bottleneck of the system is in transaction verification.

Finally we propose a new in-memory index that is also storage-friendly. A "lazy-trie" is a variant of the hash-trie data structure that achieves near-optimal height, has practical storage overhead, and can be maintained on-disk with standard write-ahead logging.

We present *CedrusDB*, a persistent key-value store based on a lazy-trie. The lazy-trie is kept on disk while made available in memory using standard memory-mapping. The lazy-trie organization in virtual memory allows CedrusDB to better leverage concurrent processing than other on-disk index schemes (LSMs, B^+-trees). CedrusDB achieves comparable or superior per-

formance to recent log-based in-memory key-value stores in mixed workloads while being able to recover quickly from failures.

BIOGRAPHICAL SKETCH

Maofan "Ted" Yin got interested in computers in his early years. While keeping it a secret from his parents, he bought a Visual Basic 6.0 programming book with all his piggy bank savings when he was in sixth grade and encountered a wonderland where he felt he should be the ruling king. With the dream of becoming a successful software engineer, he spent four challenging but fruitful years at Shanghai Jiao Tong University, during which he re-evaluated his career goal and decided to unravel the mystery behind the computer systems. After receiving his Bachelor's degree in Computer Science in 2016, he set out on an adventurous voyage for his Ph.D., guided by his dear captains Professor Emin Gün Sirer and Professor Robbert van Renesse.

Figure 1: A defective photo of Ted, taken and developed by himself.

To my family, dearest friends and the greatest enemies in my life.

ACKNOWLEDGEMENTS

In retrospect, my PhD journey started with uncertainty, went on with surprises, and finally finished up with joy and self-fulfillment. After getting lucky in my application to receive many offers, I chose Cornell for the adventure, mostly because of my academic advisors. Ask any senior PhD students, they would probably admit that the choice was largely blind and sometimes random — no one can exactly predict what will happen in the forthcoming years, and there is no medicine for regret.

I was hoping for a more hands-on experience when it all started. Like many ambitious peers who were new to the systems area but believe in their talents, I even had a dream of publishing one top conference paper every year. However, just when I came in, Professor Gün Sirer seemed to shift to a more hands-off style of advising. I was not used to it at first and became worried at times, but whenever I look back after all these years, it made a profoundly positive impact to my career development. I learned a lot from Gün. He shared the same research interest as mine, appreciated my insights, tolerated my haste or ignorance, and greatly improved my research taste. I realized doing research is not only about publishing papers, but trying to change the world, which is still the pinned tweet on his twitter now. When I first joined the social hours in the department, many people had a mysterious smile and wished me "good luck" when they heard who I would work with. But I never had complaints about Gün. He is one of the most straightforward, sincere, responsible, and professional researcher I have ever worked with, and also a patient, parent-like teacher. If I were only allowed to list one important thing that I learned from him, it would be the courage to always fearlessly try different approaches and be a hacker, without thinking too much about the effort that could be "wasted,"

even if things do not work out in the end. Throughout my entire PhD journey, we have developed a solid mutual trust, both in research and the foundation of the startup based on the Snow/Avalanche system proposed in this dissertation. I thought he was joking when he said it would be a multi-million dollar business. For most of the time you can easily tell if he is joking, but I really did not anticipate it becoming true. Gün is also one of the coolest system researchers that I have ever seen. I used "one of" because the great minds whom I would like to acknowledge next are competitive in terms of their coolness.

Since my PhD began, I always wanted to get co-advised by Professor Robbert van Renesse. Such desire developed when I realized that what I had been learning and was focused on was distributed consensus. Robbert is a guru on various hard core systems research topics, solving those fundamental problems with practical considerations. He also played a substantial role in my first major work, the Avalanche paper, and later became my advisor. I always found discussions with him inspiring and reassuring. After Gün, Kevin, and I co-founded Ava Labs startup, he took over the entire academic advisorship and had given me great suggestions when I felt overwhelmed by the split roles between research and engineering. He is a master of time management. He advises many students, but still always has time in his schedule for substantial contribution to an on-going project among many. He still codes nowadays, and can build a system from scratch faster than many of his students. Although I know most of the legends about him, I still get shocked when I witness them. I spent half of my PhD research time with Robbert and hopefully I did not disappoint him, since I always feel I am a dwarf in knowledge, thinking, and vision while speaking in front of him. Aside from research, Robbert is a good friend and full of humor. I have dreamed of playing in a music band with him one day in the holiday

party of the department, because I practice the classical guitar now and then. But I definitely need to hone my poor skills in order to seize the chance.

I would like to express my special thanks to Dr. Dahlia Malkhi. I first saw her during the first retreat of the IC3 (Initiative for Crypto-Currencies and Contracts) organization, in my senior undergraduate year. I was invited to the activity in New York City in 2016, right before the summer. She was talking about the BFT replication perspective of blockchains, which was very new at the time. My first impression of her was a knowledgeable, capable, experienced, and cool researcher. The second time we met was during a blockchain workshop hosted in Shenzhen, where we had a brief interview-like talk for my VMware internship during breakfast. I was pushed to my limit in both my research and English abilities. The questions were challenging but inspiring. We worked closely on the research later known as "HotStuff" starting in the summer of 2018. Dahlia is an excellent theoretician as well as engineer. We had a lot of meetings while at VMware Research and the discussions sometimes got a bit intense when one wanted to persuade the other. As a senior researcher and former professor, she did not overrule my arguments, but patiently listened and paid much respect. Most of the novel ideas and contributions in the final version of HotStuff were born through this kind of superproductive and cheering discussions. Gün once said "I hope you could find some peace of mind by talking to Dahlia" I always did. Although now she is busy as the CTO of Diem Association, I am sincerely looking forward to future research collaboration opportunities with her.

Another special thank goes to Professor Lorenzo Alvisi. A hidden history that few know about was that I took his distributed systems summer course while I was in my sophomore year at SJTU. He showed us that there are many interesting algorithms used in computer systems and that this area is not only

about pure engineering and coding, but also about proofs and intricate protocols. He was at UT Austin back then and probably would not have guessed that his course had more or less led me to a risky path of my PhD application, because I was doing research in AI and speech recognition throughout my undergraduate. Ironically, although some junior researchers may think that I have a decent knowledge of distributed consensus, none of them know that I got a relatively poor score in Lorenzo's summer course. I felt conflicted because these algorithms are so interesting yet I did not do well in the exam, but still deliberately chose distributed consensus as the main research direction for my entire PhD career. I chose it not because it is easy, but exactly because it is hard. I am so glad that I made this choice. Lorenzo and I also had a long talk during OSDI 2018 at Carlsbad. He comforted and also encouraged me, since the first submission of the Avalanche paper was rejected, which was my first academic paper in my PhD research. It is hard to believe that we have never had any research collaboration so far. We definitely should fix that in the future.

Other than my major research focus, I would like to thank Professor Adrian Sampson both for his CS6110 Advanced Programming Languages course and the project for my ECE minor fulfillment. The PL course let me realize my potential in resolving twisted and complicated logic and I could not believe that I got the final challenging problem right in the last exam. I almost aced the exam except for a single minor mistake. That is perhaps the last school exam I took in my life, and it taught me there is some kind of math for which I may have some potential. After all, compared to other strong PhD peers, I always doubted my math capabilities. I am more confident now. For the ECE minor, I learned a lot from building the tiny RISC-V computer system all from scratch.

I also would like to thank Professor Robert D. Kleinberg for his insights on the Avalanche project in its early stage and his service as Director of Graduate Studies, which has made our graduate life much easier.

I would like to express my gratitude towards research collaborators: Ittai Abraham, Guy Golan Gueta, Kartik Nayak, Michael K. Reiter, Ling Ren, Kevin Sekniqi and Hongbo Zhang, and my friends who had active research-related discussions with me and/or mentally supported my research: Soumya Basu, Hao Chen, Lequn Chen, Philip Daian, Cong Ding, Ayush Dubey, Yue Guo, Yiqing Hua, Xun Huang, Eric Lee, Wei-Kai Lin, Yun Liu, Sishan Long, Xuan Luo, Kai Mast, Haobin Ni, Youer Pu, Weijia Song, Zhen Wei, Zikai Wen, Siqiu Yao, Fan Zhang, Hongbo Zhang, and Yunhao Zhang. I apologize in advance that it may not be a comprehensive list and that names are in alphabetic order.

Finally, I would like to sincerely thank my parents who unconditionally support my hobby which has become my career and love me ever since I was born.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Blockchains and cryptocurrencies have attracted a lot of attention in recent years. Despite their controversial market status, the system infrastructures that support blockchains have ignited interest in revisiting some fundamental system topics such as distributed consensus and storage systems. Blockchain systems provide one or two of the main services: decentralized payments and decentralized computation, exemplified by the two most popular projects with the most circulating capital in the market: Bitcoin and Ethereum. We first take a look at these two services by a brief introduction to both projects and their history, followed by discussing their limitations and challenges. Finally, we outline our main contributions.

1.1 Decentralized Payments: Bitcoin

In 2007–2008, due to the systemic risk of capitalism, the most severe financial crisis since the Great Depression came to the market. While it caused tragic unemployment, suicides, and regression of the global economy, it also reassured some groups of activists on the Internet with their ideology that is close to anarchism and libertarianism. In an online cryptography mailing list, a hacker[1] under the pseudonym "Satoshi Nakamoto" posted a sketch (a "white paper") [119] of a futuristic, Internet-scale, online cash payment system in late 2008. In the following year, as promised by Nakamoto, "Bitcoin" was launched to maintain a universal ledger that is replicated and collaboratively supported by anonymous nodes distributed around the world. The ledger consists of a chain of blocks,

[1] or a group of people

each of which contains the hash digest of the previous block's entire content, forming a sorted, verifiable history of the contained payloads.

Bitcoin has achieved the goal of an autonomous, decentralized payment system with its own digital ("virtual") currency (BTC). Its main contribution is to address the "double-spending" problem where its predecessors have failed. Digital signatures can ensure only the owner can spend her balance, but they do not prevent the owner from deceiving a distributed system into accepting two correctly signed but conflicting transactions which both try to spend the same funds.

To model these conflicts cleanly, it proposes a model later known as Unspent Transaction Output (UTXO), which captures the transfer flow of each "coin" so that the actual balance for a user is effectively the dangling, unspent outputs from last transactions. The initial coin, named *coinbase*, is issued by the protocol to the creator of a block as the minting mechanism. Each transaction takes the unspent coins (outputs from an earlier, or coinbase transaction) as inputs and rearranges the values into its outputs.

With this model, as long as one system can guarantee the invariant that each output is consumed by any transaction as an input at most once, and this relation is consistent among all honest nodes, and the underlying digital signature primitive (an elliptic curve based algorithm, for example) holds its security guarantees, the payment system works because all transactions are verifiable and compatible. The invariant formalizes the prevention of double-spending.

Achieving the invariant, however, is non-trivial. In order to achieve *Agreement* among the honest nodes given an unknown identity set of malicious par-

ticipants under a certain size threshold, some *Byzantine Fault Tolerant* (BFT) protocol is required. Although not explicitly mentioned in the original white paper, Nakamoto proposed a simple, interesting protocol that achieves the goal similar to the standard definition of BFT consensus, but with probabilistic safety given some notion of *finality* external to the protocol itself.

In Bitcoin, in addition to the parent hash, each hash-chained block contains an extra 4-byte auxiliary field called *nonce*. Since it is not part of the actual payload (e.g., a list of user transactions), it merely offers a way to change the hash of the entire block content with the fixed payload. Then the protocol stipulates only a block whose sha256 hash value, when treated as a 256-bit unsigned integer, is smaller than a given value (i.e., prefixed by a certain numbers of zeroes) should be considered valid. Because of the pseudo-randomness of the cryptographically strong sha256 hash algorithm, the only known method to find out a proper nonce that leads to the eligible hash value is to grind through the search space of the nonce. Therefore, a *Proof-of-Work* (PoW) process is required to enumerate the nonce, which is also referred to as *mining* by the community, in an analogy that it takes a node ("miner") some substantial amount of computation and power to generate a valid block, which in turn rewards the node with a coinbase transaction. Bitcoin dynamically determines the mining difficulty by the average mining time in a window of recent blocks [150].

The Proof-of-Work idea was not new even in 2008: Bitcoin uses a similar PoW to that proposed by Adam Back in 2002 [20], predated by an attempt of issuing token money with a Reusable Proof-of-Work (RPoW) [75]. The credit of the first PoW goes to the effort of fighting against e-mail spams [70].

The idea of using nested hashes (in chained blocks) to have an immutable history of records was not new either. Among the references of the white paper, there is one work [23] that first mentioned this idea and the authors had even put their scheme into action by advertising their timestamping service notice weekly in the classified section of *The New York Times* since 1995.

What Bitcoin has contributed is an insightful approach that combines these two ideas into a BFT consensus protocol. When each block carries a PoW whose difficulty is properly throttled by the computational capability of the combined miners, the *block rate* (i.e. the number of blocks all nodes can generate in a unit of time) remains controlled solely by the protocol, because the miner can only grind through the nonces, regardless of its intention. Moreover, Bitcoin uses a rule that nodes should always append their new blocks to the longest local chain (*fork*) and deem that longest chain as the canonical ledger. By a practical and proper assumption of the time that it takes to disseminate and synchronize the blocks, closely related to the overall PoW time (*difficulty*) [123], the system will eventually converge on an ever growing canonical chain whose tip may flip around at times, but the prefix becomes exponentially difficult to be altered by the effort of malicious (*Byzantine*) nodes, as its depth from the tip increases.

With this computationally expensive yet powerful approach, Bitcoin nodes keep mining all the time and resolve the conflicting transactions according to the UTXO model by rejecting those who come later in the topological order of blocks on a chain, a path in the tree of all generated blocks. The miner can also easily guarantee that transactions in the same block do not conflict with each other.

1.2 Decentralized Computation: Ethereum

Inspired by Bitcoin's blockchain, Vitalik Buterin proposed the idea of utilizing its ordering capability to host general computations in 2013. Ethereum [152] was later built with its own abstraction by a collaborative effort from its community. Unlike Bitcoin which "hard-codes" its transactions with the UTXO model directly as block payloads, Ethereum further utilizes the more generalized state machine replication ability enabled by the implicit, probabilistic BFT consensus.

Ethereum defines a computation model named *smart contract*, where each contract is an object with persistent, on-chain state and can be interacted via a "call" to one of its methods in a transaction. The methods are written in high-level programming languages such as Solidity (Javascript-like) or Vyper (Python-like) by the user and get compiled down to the bytecode for the Ethereum Virtual Machine (EVM). Each method acts as a potential state observation or transition of the smart contract and may query or update its persistent state. Like a user account, a smart contract also has its own balance in Ethereum's virtual currency (ETH), which is transferable and kept as part of its state.

In short, Ethereum supports its decentralized "world computer" by encoding all state changes as transactions. To deploy a smart contract, the transaction carrying the bytecode will be sealed in a block that is later part of the canonical chain. To run the code, one just needs to construct a transaction with the arguments for the invoked method of the targeted smart contract (the "contract ABI") and put it on-chain as well. The chain thus encodes both the I/O and the logic of the computation like the trace of an operating system, while the lo-

cal program of the node can replay the trace by emulating EVM instructions to obtain the state at any given point in the history.

1.3 Performance and Scalability Challenges

Both Bitcoin and Ethereum use chained PoW to support their blockchain services, which is known as *Nakamoto consensus* [25,73,78,91,119,123–125,143–145]. Nakamoto consensus is easy to implement and robust, and the PoW also naturally serves as an admission control so the system is safe as long as less than half of the computation power is adversarial (the attack that breaks this assumption is known as a "51% attack"). It does not require accurate membership knowledge like classical BFT consensus protocols and does not need explicit reconfiguration when nodes join or leave. These advantages had lowered the barrier of implementing early blockchain projects who followed Bitcoin's path.

However, as more people become interested in harnessing the power of blockchain decentralization and attempt to replace the traditional centralized Internet/financial services, the performance gap has become a major deal breaker. For example, although Bitcoin can serve as an online cash payment system, it serves only around 3 transactions [30] per second for the entire world. To reach the recommended safety, there is a "six-block" rule that is widely adopted by exchanges and users, whereas each block takes around 10 minutes [29] to generate in the network. Therefore, the end-to-end latency for considering a transaction as confirmed usually takes around an hour, which is bizarrely high compared to the competing services that use databases and centralized servers such as Paypal or Alipay [83]. Ethereum has a shorter block time at around 15

seconds [72]. But to reach the same level of safety, one has to wait for more blocks that bury the block that contains the submitted transaction. It is recommended that one need to wait for around 40 blocks to match the six-block rule in Bitcoin. Waiting around 10 minutes for the confirmation is still too long. Plus, the throughput of Ethereum is only around 15 transactions per second, still way too low compared to those centralized services. The underlying Nakamoto consensus also cannot quiesce by construction: their security relies on constant participation by miners, even when there are no decisions to be made. It is not green, either: Bitcoin currently consumes around 76 TWh/year [63], more than twice as all of Denmark [45]. The inherent scalability bottleneck is difficult to overcome through simple reparameterization [55].

There have been extensive studies on BFT consensus protocols since 1970, but the area ceased to be active after reaching its peak by 2010. We believe that the reason lies with the adoption of Crash Fault Tolerant (CFT) consensus protocols, in IT industry. At around 2000, Internet companies like Google started to build their infrastructures [37], with the help of virtualization technology for what was later known as "The Cloud" [51]. The industry became interested in an algorithmic solution that can be used to tolerate failures in a small group of machines (e.g., Google used commodity-level machines in its early days). Failures are inevitable for any hardware and the most common type of failures is a crash where a machine no longer responds to the network. In a CFT consensus protocol such as Paxos [95] or Raft [122], there are typically 3–5 nodes participating in voting and replicating the state, where the crash of any 1–2 nodes will not affect the consistency of the states among the others. Thus, the entire cohort can still exhibit a correct state to the external environment, masking the failures. BFT consensus, compared to CFT consensus, has a weaker assumption

about the participants, allowing some threshold of them to disobey the protocol and even collude to overthrow the system. These protocols inherently consume more resources and are significantly slower than their CFT counterparts. They also tolerate fewer number of faults given the same number of nodes. While they tolerate malicious behavior, this is not really needed in a typical industrial setup where all machines are managed by the same entity, behind firewalls and other security measures.

After blockchain got its attention, researchers familiar with BFT consensus protocols regained interest in such classical protocols. These protocols, unlike Nakamoto consensus, have different assumptions but a similar goal. They require common knowledge of all participants and always guarantee safety (deterministically, not probabilistically as in Nakamoto's) and finality as long as the assumptions are held. The tolerance of adversarial nodes is some threshold in terms of the number of nodes, rather than the computational power. They do not require time-consuming, "useless" PoW computation, but operate by voting. With reconfiguration techniques (which use consensus to change the parameters of the protocol itself) and some admission rules (by economic arguments such as Proof-of-Stake, or real-world authorization to make it "permissioned"), these protocols can be much more efficient than Nakamoto consensus, while still being decentralized.

Although these protocols are efficient for a small number of nodes (typically from four up to several dozens of nodes), they scale poorly to hundreds or thousands. Their major bottleneck is in handling the communication according to the algorithm's logic. There is usually at least one node (replica) that needs to broadcast messages to the entire system. Or even worse, in many protocols,

there is at least one phase of execution where *every* node needs to broadcast messages, resulting in quadratic, or sometimes even cubic cost in communication. Besides, compared to Nakamoto consensus, practical BFT consensus protocols (e.g. PBFT) are notoriously difficult to comprehend and implement, even for the experts. They usually consist of a fast path where the system logic is straightforward and a slow path (such as a *view change*) where failures are handled, and both paths are intertwined.

Finally, apart from BFT consensus protocols, we believe there is an emerging challenge that could be the very next bottleneck for blockchain systems when consensus bottlenecks have been mitigated to some extent. While the consensus protocol provides a fault-tolerant and consistent ordering of data, a blockchain system still needs to persist its operating state and schedule the access of the persisted state from the memory or the disk efficiently. In recent years, according to the author's own experience and the complaints heard from other well-known projects, the storage system, which is usually implemented by a *key-value store*, can become the new bottleneck, especially for Ethereum-like systems that allow complex user transactions and state machine execution (e.g., EVM and smart contracts).

1.4 Solutions

Given the aforementioned major challenges in either consensus scheme, we took a step back and explored new protocols designed from scratch. We designed two novel consensus protocols: "HotStuff" and "Snow," targeted for different applications. Based on Snow, we built an Internet-scale payment system

named "Avalanche." Finally, we took a stab at improving the storage system for blockchains and other applications. We designed and built a fully functional key-value store, "CedrusDB," that achieves comparable or superior performance to recent log-based in-memory key-value stores in mixed workloads while being able to recover quickly from failures.

1.4.1 HotStuff

Since the introduction of PBFT [43], the first practical BFT replication solution in the partial synchrony model, numerous BFT solutions were built around its core two-phase design. The practical aspect is that a stable leader can drive a consensus decision in just two rounds of message exchanges. The first phase guarantees proposal uniqueness through the formation of a quorum certificate (QC) consisting of $n - f$ votes (where n is the number of processes and f is the maximum number of processes that are allowed to fail). The second phase guarantees that the next leader can convince replicas to vote for a safe proposal.

The algorithm for a new leader to collect information and propose it to replicas—called a *view-change*—is the epicenter of failure recovery. Unfortunately, view-change in the two-phase design is far from simple [112], is bug-prone [6], and incurs a significant communication penalty for even moderate system sizes. It requires the new leader to relay information from $n - f$ replicas, each reporting its own highest known QC. Even counting just authenticators (digital signatures or message authentication codes), conveying a new proposal has a communication footprint of $O(n^3)$ authenticators in PBFT, and variants that combine multiple authenticators into one via threshold digital signatures

(e.g., [40,80]) still send $O(n^2)$ authenticators. The total number of authenticators transmitted if $O(n)$ view-changes occur before a single consensus decision is reached is $O(n^4)$ in PBFT, and even with threshold signatures is $O(n^3)$. This scaling challenge plagues not only PBFT, but many other protocols developed since then, e.g., Prime [14], Zyzzyva [92], Upright [49], BFT-SMaRt [26], 700BFT [19], and SBFT [80].

We propose HotStuff, a protocol that revolves around a three-phase core, allowing a new leader to simply pick the highest QC it knows of. It introduces a second phase that allows replicas to "change their mind" after voting in the phase, without requiring a leader proof at all. This alleviates the above complexity, and at the same time considerably simplifies the leader replacement protocol. Last, having (almost) canonicalized all the phases, it is easy to pipeline HotStuff, and to frequently rotate leaders.

To the best of our knowledge, only BFT protocols in the blockchain arena like Tendermint [35, 36] and Casper [38] follow such a simple leader regime. However, these systems are built using synchronous assumptions, wherein proposals are made in pre-determined intervals that must accommodate the worst-case time it takes to propagate messages over a wide-area peer-to-peer gossip network. In doing so, they forego a hallmark of most practical SMR solutions (including those listed above), namely *optimistic responsiveness* [125]. Informally, *responsiveness* requires that a non-faulty leader, once designated, can drive the protocol to consensus in time depending only on the *actual* message delays, independent of any known upper bound on message transmission delays [18]. Optimistic responsiveness, which is more appropriate for our model, requires responsiveness only in beneficial (and hopefully common) circumstances—

here, after Global Stabilization Time (GST) is reached. Optimistic or not, responsiveness is precluded with designs such as Tendermint/Casper. The crux of the difficulty is that there may exist an honest replica that has the highest QC, but the leader does not know about it. One can build scenarios where this prevents progress ad infinitum (see Section 3.2.4 for a detailed liveless scenario). Indeed, failing to incorporate necessary delays at crucial protocol steps can result in losing liveness outright, as has been reported in several existing deployments [3,4,41]. HotStuff operates by only waiting for messages with their actual delays and thus it achieves optimistic responsiveness.

1.4.2 Snow

Inspired by gossip algorithms, we propose the Snow consensus protocol family, which gains its properties through a deliberately metastable mechanism. Specifically, the system operates by repeatedly sampling the network at random, and steering correct nodes towards a common outcome. Analysis shows that this metastable mechanism is powerful: it can move a large network to an irreversible state quickly, where the irreversibility implies that a sufficiently large portion of the network has accepted a proposal and a conflicting proposal will not be accepted with any higher than negligible (ε) probability.

Similar to Nakamoto consensus, the Snow protocol family provides a probabilistic safety guarantee, using a tunable security parameter that can render the possibility of a consensus failure arbitrarily small. Unlike Nakamoto consensus, the protocols are green, quiescent and efficient; they do not rely on PoW and do not consume energy when there are no decisions to be made. The effi-

ciency of the protocols stems partly from removing the leader bottleneck: each node requires $O(1)$ communication overhead per round and $O(\log n)$ rounds in expectation, whereas classical consensus protocols have one or more nodes that require $O(n)$ communication per round (phase). Furthermore, the Snow family allows discrepancies in knowledge of membership, as discussed in [136]. In contrast, classical consensus protocols require full and accurate knowledge of membership as its safety foundation.

Snow's subsampled voting mechanism has an additional property that improves on previous approaches for consensus. Whereas the safety of quorum-based approaches breaks down immediately when the predetermined threshold f is exceeded, Snow's probabilistic safety guarantee degrades smoothly when the number of Byzantine participants exceeds f.

1.4.3 Avalanche

To demonstrate the potential of the Snow protocol family, we illustrate a practical peer-to-peer payment system, *Avalanche*. Avalanche is based on "Snowball," a consensus protocol in the Snow protocol family. In effect, Avalanche executes multiple Snowball instances with the aid of a Directed Acyclic Graph (DAG). The DAG serves to aggregate multiple instances, reducing the cost from $O(\log n)$ to $O(1)$ per node and streamlining the path when there are no conflicting transactions.

Avalanche is currently deployed and in active use worldwide. There are more than 900 nodes (>800 of those publicly owned) directly participating in the protocol confirming over 20K transactions every day, with around 3 billion

dollars worth of market capital in total circulation. The open system has been up since September 2020, allowing anyone to join using open-source code.

1.4.4 CedrusDB

In this thesis, we propose *lazy-trie*, a storage-friendly data structure. All nodes in a lazy-trie have the same number of children slots, simplifying maintenance. To bound the depth of the trie and probabilistically balance the load, user keys are hashed to index into the trie. To further reduce the depth of the trie and improve utilization, the lazy-trie uses a path compression technique similar to radix trees [118]. Finally, some small subtrees are collapsed into linked lists at leaves, greatly reducing storage overhead and read/write amplification.

We use the lazy-trie data structure to implement a memory-mapped, persistent key-value store, *CedrusDB*. It is able to achieve near-optimal dynamic tree height with practical storage overhead. Like LMDB [53], the implementation of CedrusDB uses memory-mapping, but, unlike LMDB, CedrusDB does not require that all of the data set fit in available memory—CedrusDB implements its own page replacement and does not rely on the operating system kernel to do so. The lazy-trie organization in virtual memory allows CedrusDB to better leverage concurrency.

A shortcoming that CedrusDB shares with FASTER is that hashing makes support for range queries difficult. Fortunately, not all applications require support for range queries and there are various proprietary and open source key-value stores that do not support them [47, 60, 93, 104, 106]. Many applications use a key-value store to persist user data by keys. For example, a blockchain

application stores data using keys that are already hashes. The Shadowfax distributed key-value store based on FASTER, which is unordered and persistent like CedrusDB, serves 130 Mops/s/VM in the Microsoft Azure cloud [93]. There also exist practical techniques for supporting range queries on top of key-value stores that do not [127].

1.5 Contributions

In summary, this dissertation makes the following contributions:

- We design and implement HotStuff. To the best of our knowledge, HotStuff is the first partially synchronous BFT SMR (consensus) protocol that achieves the following two properties:
  - **Linear View Change**: After GST, any correct leader, once designated, sends only $O(n)$ authenticators to drive a consensus decision. This includes the case where a leader is replaced. Consequently, communication costs to reach consensus after GST is $O(n^2)$ authenticators in the worst case of cascading leader failures. See Table 1.1 for the comparison to other protocols.
  - **Optimistic Responsiveness**: After GST, any correct leader, once designated, needs to wait just for the first $n - f$ responses to guarantee that it can create a proposal that will make progress. This includes the case where a leader is replaced.

Protocol	Authenticator complexity			Responsiveness
	Correct leader	*Leader failure (view-change)*	*f leader failures*	
DLS [69]	$O(n^4)$	$O(n^4)$	$O(n^4)$	
PBFT [43]	$O(n^2)$	$O(n^3)$	$O(fn^3)$	✓
SBFT [80]	$O(n)$	$O(n^2)$	$O(fn^2)$	✓
Tendermint [35] / Casper [38]	$O(n^2)$	$O(n^2)$	$O(fn^2)$	
Tendermint* / Casper*	$O(n)$	$O(n)$	$O(fn)$	
HotStuff	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(fn)}$	✓

*Signatures can be combined using threshold signatures, though this optimization is not mentioned in their original works.

Table 1.1: Performance of selected protocols after GST.

- In HotStuff, the costs for a new leader to drive the protocol to consensus is no greater than that for the current leader. As such, HotStuff supports frequent succession of leaders, which has been argued is useful in blockchain contexts for ensuring chain quality [78].
- HotStuff achieves its properties by adding another phase to each view, a small price to latency in return for considerably simplifying the leader replacement protocol. This exchange incurs only the actual network delays, which are typically far smaller than Δ in practice. As such, we expect this added latency to be much smaller than that incurred by previous protocols that forgo responsiveness to achieve linear view-change. Furthermore, throughput is not affected due to the efficient pipeline we introduce in Section 3.3.
- In addition to the theoretical contribution, HotStuff also provides insights in understanding BFT replication in general and instantiating the protocol in practice (see Section 3.4):

  - A framework for BFT replication over graphs of nodes. Safety is specified via voting and commit graph rules. Liveness is specified separately via a *Pacemaker* that extends the graph with new nodes.
  - A casting of several known protocols (DLS, PBFT, Tendermint, and Casper) and one new (ours, HotStuff), in this framework.

- HotStuff has the additional benefit of being remarkably simple, owing in part to its economy of mechanism: There are only two message types and simple rules to determine how a replica treats each. Safety is specified via voting and commit rules over graphs of nodes. The mechanisms needed to achieve liveness are encapsulated within Pacemaker, cleanly separated from the mechanisms needed for safety. At the same time, it is expressive in that it allows the representation of several known protocols (DLS, PBFT, Tendermint, and Casper) as minor variations. In part this flexibility derives from its operation over a graph of nodes, in a way that forms a bridge between classical BFT foundations and modern blockchains.

- We describe a prototype implementation and a preliminary evaluation of HotStuff. Deployed over a network with over a hundred replicas, HotStuff achieves throughput and latency comparable to, and sometimes exceeding, those of mature systems such as BFT-SMaRt, whose code complexity far exceeds that of HotStuff. We further demonstrate that the communication footprint of HotStuff remains constant in face of frequent leader replacements, whereas BFT-SMaRt grows quadratically with the number of replicas.

- We introduce a family of binary BFT protocols, Snow, built around a metastable mechanism via network subsampling. The protocols offer strong probabilistic safety and operate without a leader. Every node samples votes in each round, without polling the entire network, making the protocols fun-

damentally scalable. Without the need of PoW, they are also quiescent and green in the fashion of classical protocols. But unlike those, nodes only need to process a constant number of votes per round, and go through logarithmic expected number of rounds in total. Also by sampling, they do not require accurate knowledge of the membership and the votes are plain messages from the sender, so no signatures (and no QCs) are required by the consensus, after the channel is authenticated during the setup.

- We implement the core of an Internet-scale, electronic payment system, Avalanche, based on the Snowball protocol. Avalanche is evaluated in a deployment with thousands of nodes all participating in the agreement process. The deployment can still maintain high throughput, low confirmation latency and outperforms other decentralized payment systems.

- We design the lazy-trie data structure, an index that is friendly to both in-memory and on-disk access. With a practical storage footprint, it dynamically grows with near-optimal tree height and allows for efficient concurrent access.

- We design and implement CedrusDB based on lazy-trie. The memory-mapped lazy-trie allows CedrusDB to achieve better performance in read operations and concurrent access than other on-disk index schemes. It also recovers much faster than recent key-value stores that use in-memory indexes. We evaluate CedrusDB in mixed workloads and it achieves comparable or superior performance to its main competitors.

CHAPTER 2

BACKGROUND

2.1 Time, Clocks and Ordering

One major characteristic that makes designing distributed systems protocols (algorithms) hard is *asynchrony*. In a distributed system, participating nodes (aka. processes) talk to each other via some communication channels. The unpredictable delay in messaging and event propagation results in the absence of a reliable *time* definition. While nodes can have their own local clocks that are periodically synchronized to the real time, the clocks experience drift in reality, where the asynchrony creeps in again. Lamport's seminal paper [94] talks about how to create a well-defined notion of time based on either logical or real-time clocks. The main reason for having time in a system is to establish ordering of events that happen asynchronously across the entire system. The paper is famous for the proposed "Lamport clock," and is also the first paper that sketches the general abstraction that uses a *state machine* and a total ordering algorithm to describe any distributed systems.

2.2 Fault Tolerance, State Machine Replication and Consensus

Similar to asynchrony, failures are also common for real-world systems. They usually result from the limitations of human engineering, the deterioration of hardware materials, or even the evil desire of human minds. A fault can be caused by a crashing program, a failing chip, a power outage, or even a hacker's

infiltration. According to the deployment environment and condition, faults differ in their likelihoods and severity. A typical categorization is as follows:

Fail-stop. When a node is faulty, it stops its normal execution upon the failure, whereas other non-faulty nodes are eventually aware of its faultiness. A faulty node can be viewed as sending out a "dying message" right before its failure. The ability of being able to eventually sense the faulty nodes can also be abstracted by a perfect *failure detector* [46]. From the network model perspective, this type of failure implies a *synchronous* system. As an example, Chain Replication [148] is fault tolerant under the fail-stop failure model.

Crash. A faulty node silently stops its normal execution. The other nodes may or may not be aware of the failure. The key difference between crash and fail-stop is that in crash model, one cannot tell whether the irresponsiveness of a node is due to its crashing or merely the network delay. It is indistinguishable because of the underlying network asynchrony. Both Paxos and Raft are crash fault tolerant protocols.

Byzantine. In this case, a faulty node can crash as in the crash model or exhibit arbitrary behavior by choosing to send or receive any messages. It is the weakest assumption as it basically makes no assumption on the node's protocol behavior. The adversarial nodes may even pretend to be correct until some favorable moment when they launch a coordinated attack to the system by their own strategy. The proposed protocols in this dissertation tolerate this kind of failure.

In order to mask failures and improve availability of the system, we would like to consider having the aforementioned state machine operated with an ordering algorithm that is fault tolerant. Here we are interested in tolerating faults with software, which is usually done by introducing redundancy into the system, such as replication. In *State Machine Replication* (SMR), a distributed system consists of *replica* nodes, or replicas, that all try to maintain the state transitions of the same targeted state machine. A concrete example of the state machine could be a database system where each transaction (or operation) is a potential state transition, whereas the persistent indexes and data kept by the database are the state. In a fault-tolerant, distributed database system, each replica keeps the entire storage state by applying the totally ordered state transitions, determined by the SMR protocol. More formally, an SMR protocol typically guarantees the following, in the presence of some faulty replicas:

- **Agreement.** On any two non-faulty replicas r and r' (r' can be r) whose sequences of *committed* (or "decided," "executed") operations are l and l' at any respective local time, l is a prefix of l' or the other way around.
- **Termination.** Each client operation submitted to the SMR protocol should be eventually committed by every non-faulty replicas.

An SMR protocol can be built from an equivalent but more basic *consensus* protocol. In a consensus protocol, all processes (i.e., replicas) initially propose a value given by its input, and the protocol typically guarantees:

- **Agreement.** For non-faulty processes that decide, they agree on the same value.

- **Integrity.** If all non-faulty processes propose the same value, then if a process decides it decides that value.
- **Termination.** Every non-faulty process eventually decides some value.

For a consensus protocol, the *integrity* property is sometimes also referred to as "validity" or "non-triviality." In practice, consensus protocols (even Byzantine tolerant ones) may not care about whether the decided value is proposed by a faulty process, assuming the value proposed by any process is always *valid* by some input validation. The integrity here merely serves to prevent a trivial solution that always decides a hard-coded value.

With a consensus protocol, one can derive an SMR protocol by treating the operation together with its sequence position in the state machine execution as a single value and run a consensus protocol for each position in the sequence, one after another. To implement consensus with SMR is trivial. Therefore, from now on, to extent of this dissertation, we use both terms interchangeably because of their equivalence.

There is another kind of protocols that is closely related to consensus called *Reliable Broadcast* (RB). RB usually has a sender (leader) and several receivers (replicas), where the sender has a special role in the protocol that it initially broadcasts a message (value) to the receivers by its input. Each non-faulty receiver may deliver (output) some value. Depending on the failure model (and the network model), its definition varies in a subtle way. For example, a *Terminating Reliable Broadcast* has close relation to consensus with crash fault tolerance, and it has a special deliverable value called "sender failure." *Byzantine Reliable Broadcast* (BRB) was first formulated by Bracha [33] (and known as Bracha's protocol). Typically, a BRB protocol guarantees the following:

- **Agreement.** If two or more non-faulty receivers deliver values, then they deliver the same value.
- **Validity.** If the sender is non-faulty and a receiver delivers a value, then the value was sent by the sender.
- **Termination.** If the sender is correct and sent a value, or if a correct receiver delivered a value, then eventually all correct receivers deliver a value.

(B)RB is weaker than consensus by its definition because the sender has a special role of input, while in consensus, all processes get their inputs. It is usually used as a powerful primitive in crafting a consensus protocol. Notably, most practical consensus (SMR) protocols are not built from (B)RB, but directly designed from scratch.

2.3 Impossibility and Beyond the Impossibility

2.3.1 FLP's "Verdict"

Unfortunately, the consensus problem, by its standard definition, is unsolvable in a fully asynchronous network. In an asynchronous network, messages can take arbitrarily long to deliver to the receiver. We assume all network channels between correct processes are reliable in that messages will not be dropped, duplicated, or corrupted during their transimission.

In the early years, as researchers and engineers tried to solve the consensus problem in an asynchronous network, they found that there seemed to be

a dilemma in achieving both safety (agreement) and liveness (termination). In 1985, Fischer, Lynch and Paterson confirmed this folklore by proving that the consensus problem is impossible to solve for an asynchronous system with failures [76]. The proof models any deterministic consensus protocol as an automata and describes possible execution traces of the entire system by schedules of the configurations. They proved by contradiction that for any consensus protocol, there always exists a non-terminating path of its execution.

The impossibility, however, does not prevent people from designing practical consensus protocols. One can still get around the impossibility by changing some assumptions to weaken the problem:

- **Assume a synchronous network.** The first Byzantine fault tolerant protocols proposed in the famous Byzantine general's problem [97] paper are synchronous protocols. A *synchronous consensus* protocol assumes there is a strict and known upper bound of the message delay for all channels.
- **Use probabilistic termination.** One can replace the deterministic termination requirement with a probabilistic one. While FLP's result indicates there exists an infinite undecidable execution path, the path can have an exponentially small probability as it extends. In other words, it is *possible* that one keeps getting a head face when tossing a coin, but getting an infinite outcome of all heads has a probability of 0. An *asynchronous probabilistic consensus* protocol guarantees a probability of 1 for termination.
- **Terminate only in presence of synchrony.** A practical way of solving consensus is to be able to terminate with some limited rounds of message exchange when the network is "good," while the agreement property is still guaranteed even if the network is entirely asynchronous. Another slightly

different perspective is to not rely on the known latency bound for the agreement even when it exists, so the system is still synchronous but the bound is just unknown. By taking either approach, these protocols are called *partially synchronous* protocols.

2.3.2 Synchronous Consensus

The upper bound of message delivery is denoted by Δ. Protocols usually operates in rounds of full synchronization, where each round has bounded time as no non-faulty processes go out-of-sync more than Δ. The "lockstep" scheme of synchronization makes it easy to prove correctness and also provides strong guarantee in timing. The protocols thus have predictable, bounded time for termination, even in presence of Byzantine participants. On the other hand, the strong synchronization creates bottleneck in throughput. The throughput depends on the choice of Δ, which cannot be too large for a high throughput, but not too small to be impractical or fragile. Thus, synchronous consensus protocols may be more suitable for real-time, embedded applications where the delay can be mostly guaranteed by hardware or circuits and response time is expected to be bounded, such as aircrafts, motor vehicles or other industrial control systems. There are recent advances that try to remove the lockstep style of execution and improve the performance [10, 82].

2.3.3 Asynchronous Probabilistic Consensus

Ben-Or proposed the first theoretical BFT consensus protocol that utilizes randomization to guarantee probabilistic termination, in an asynchronous network. Because asynchronous probabilistic consensus protocols terminate with a stochastic process, they usually require some number of rounds in order to converge to the final decision. The expected number of rounds required by Ben-Or grows exponentially as the number of participants grows (assuming the maximum number of failures grows proportionally with the number of participants). Many protocols are also built from BRB. Bracha demonstrated how to achieve only a logarithmic expected number of rounds [34]. Another metric for complexity is the total number of messages (or authenticators) exchanged in expectation. The complexity of HoneyBadger [115], a protocol designed for blockchains, is $\Omega(n^3)$, while the latest known result, Validated Asynchronous Byzantine Agreement [11] requires $O(n^2)$, inspired by the complexity reduction of HotStuff in this dissertation. Many "Proof-of-Stake" (PoS) protocols used by blockchain projects are also asynchronous BFT protocols, as we'll later introduce. This kind of protocol utilizes randomization (e.g., a coin-flipping algorithm) to fulfill liveness and drive the convergence of the agreement from a split in inputs. Unlike partially synchronous protocols, they usually do not have the notion of a "leader" (or a "strong leader") who dictates the next proposal. A common misunderstanding is that since these protocols do not require a superficial "leader," they scale better. In fact, asynchronous protocols always require all-to-all state dissemination, where every process endures the processing load equivalent to that of a leader in partially synchronous protocols and thus they are strictly more expensive than their partially synchronous counterparts.

2.3.4 Partially Synchronous Consensus

Because asynchronous probabilistic consensus only guarantees termination in expected number of rounds of communication, from a practical stand point, one would like a protocol that only needs a deterministic, or even constant number of rounds for a faster decision. While this is not strictly possible due to FLP's impossibility, it can still be approximated in practice. Under most network setups (LAN or WAN), there is indeed an implicit upper bound of the message delay: on a "good" day, the network is not overwhelmed or congested and so it exhibits some degree of synchrony.

Given this idea, a common way to theoretically model its characteristics is to assume there is some unknown Global Stabilization Time (GST) after which the network becomes synchronous. The key difference from a synchronous network model is that in a partially synchronous network, GST is hidden and thus cannot be relied on for a protocol's safety. Therefore, this model still preserves the level of safety guarantee as in the asynchronous model, while it permits the progress by assuming the underlying network eventually becomes synchronous.

Well-known Crash Fault Tolerant SMR protocols such as Paxos and Raft also fall into this category. In Raft, heartbeats are used as a way to finally achieve the synchrony, which is not required to guarantee the agreement reached by the consensus. In 1988, Dwork, Lynch and Stockmeyer explicitly proposed this model leading to a BFT consensus solution later referred to as "DLS." This protocol preserves safety during asynchronous periods, and, after the system becomes synchronous, DLS guarantees termination. Once synchrony is maintained, DLS incurs $O(n^4)$ total communication and $O(n)$ rounds per decision. In 1999, the first pratical BFT protocol for SMR, PBFT was proposed, whose sta-

ble leader requires $O(n^2)$ communication and two round-trips per decision, and the leader replacement protocol incurs $O(n^3)$ communication. Similar to Paxos, PBFT introduced the notion of a leader to BFT consensus, which accelerates the decision process to only two round-trips of voting in its normal case.

Another way is to assume an unknown bound on network and processing latency. Protocols like Paxos and Raft can guarantee termination with such an assumption simply by continually increasing timeouts for leader election.

2.4 Paradigms: How to Make a Consensus Protocol

In this section, we briefly walk through the techniques and core ingredients that are typically used in designing a consensus (or SMR) protocol, informally.

2.4.1 Quorums vs. Nakamoto

Before the birth of Bitcoin, the research in both academia and industry used *quorums* to achieve fault tolerance. A quorum is just a certain subset or any threshold subset of the universal set of a system's participants. For a quorum-based (or voting-based) consensus protocol, processes exchange and amend their knowledge of proposals by sending votes. In a Byzantine setting, votes are usually cryptographically signed by digital signatures or authenticated using a Message Authentication Code (MAC). In a nutshell, such a system repeatedly uses the intersections of quorums (the "Pigeonhole Principle" guarantees a non-empty intersection) to reason about the invariants that need to be held by the non-faulty processes through failures.

On the other hand, Satoshi Nakamoto's Bitcoin later gave a different way of achieving an agreement. Miners collectively "vote" on what is the longest chain and which blocks to attach to the longest chain to progressively and stochastically secure the common prefix of the chain they work on. It is computationally far less efficient and its latency is inherently bottlenecked by the design, compared to quorum-based protocols. But its admission control is very different from traditional consensus protocols. For example, its safety is determined in terms of the aggregated computational power rather than the actual number of processes, making it a natural fit for an open system (i.e., a system where membership is open to all).

All of the BFT consensus protocols mentioned above are based on either one or a hybrid mix of the quorum-based and Nakamoto consensus paradigms. Our proposed Snow protocol family in this dissertation could be considered a new family that shares some commonality with both kinds, but offers a novel design direction.

2.4.2 Creating Skewness

The goal of a consensus protocol is to reach an agreement on a decided value chosen from the various input candidates. The replication system consists of replicas that are often equipped with the same hardware resources and are interchangeable. In an ideal system, ignoring the effects from networking and scheduling, each non-faulty replica behaves according to the same algorithm, making the system largely symmetric. Therefore, to reach an agreement, the system has to break such symmetry and finally collapse to one preferred value

among all inputs. There are several ways to introduce "skewness" into the system:

- **Use a Leader.** Protocols may temporarily designate a special leader role to one replica in the system. The core of consensus then effectively becomes the leader transition (known as a "view-change"). These protocols are mostly partially synchronous and fundamentally utilize synchrony (such as timeouts) and/or leader schedule to break the symmetry of leadership. Once a new non-faulty leader becomes incumbent, it can dictate the next proposals, which largely reduces the contention caused by different proposals and offers a fast decision process. Examples are: Paxos, Viewstamped Replication [121], Raft, PBFT, Zyzzyva [92], HotStuff, etc.
- **Local Randomization.** Asynchronous probabilistic protocols like Ben-Or's can simply use local random source to break the symmetry. The downside of having local randomness is the protocol may have to retry many times until a favorable candidate is adopted by the majority.
- **Global Randomization.** There are various recent asynchronous protocols such as HoneyBadger, Ouroboros [89], and Algorand (BA^*) [79] that use advanced cryptographic primitives for randomness with inter-replica correlation. This approach can be viewed as flipping a common coin, scaling well with the number of replicas.
- **Network Dissemination.** The symmetry can also be broken by fluctuations in message propagation latencies. Stellar [110] can be considered of this kind, and the Snow protocol family introduced later in this dissertation uses a gossip-like process where each node collects preferences by sampling and up-

dates its current proposal according to the aggregate mass so a discrepancy in support can be amplified and triggers a convergence.

2.4.3 Surviving Failures

A key functionality that consensus provides is fault tolerance. Protocols typically use one of the following two schemes to carry on the agreement process during failures:

- **Log-based (View-based)** SMR Protocols can sequence multiple decisions as a log and the consensus protocol determines its correct state by the longest or most up-to-date version of the log stored by some non-faulty replicas. Different versions preserved on replicas are the different views of the log and failures trigger view-changes to bump up the view number to invalidate the stale portion of the log. The participants vote on the prefix of the log and switch to the correct prefix after the view-change. Raft, Zab, Viewstamped Replication, PBFT, Zyzzyva are some popular view-based consensus protocols. Nakamoto consensus and HotStuff can also be considered in this category.
- **Slot-based (Ballot-based)** There are many SMR protocols that are constructed by multiple instances of a single-shot ("single-decree") consensus. Usually this type of protocol is closely related to Paxos. Instead of reasoning about a log prefix or a fork, this class of protocols makes decisions for individual slots in the final replicated sequence where each slot is treated by a separate consensus instance, maintaining its *ballots*. A ballot is a monotonically increased value to make sure replicas respect the historical progress (e.g., the completion of an earlier phase) for the slot in the presence of failures. The replicas

may need to fill in the gaps with no-ops for those proposals lost in failures and must only execute up to the last slot in the continuity of decided ones. Stellar is a ballot-based protocol.

2.5 Sybil Prevention: Not Really Consensus's Business

Consensus protocols provide their guarantees based on assumptions that only a fraction of participants are adversarial. These bounds could be violated if the network is naively left open to arbitrary participants. In particular, a Sybil attack [67], wherein a large number of identities are generated by an adversary, could be used to exceed the bounds.

A long line of work, including all aforementioned quorum-based protocols, treats the Sybil problem separately from consensus, and rightfully so, as Sybil control mechanisms are distinct from the underlying agreement protocol[1]. In fact, to the best of our knowledge, only Nakamoto-style consensus has "baked-in" Sybil prevention as part of its consensus, made possible by PoW, which requires miners to continuously stake a hardware investment. Other protocols may rely on Proof-of-Stake (by economic argument), Proof-of-Authority (by administrative argument that makes the system "permissioned"), or Proof-of-Space (another kind of economic argument that requires stacking some storage devices). The consensus protocols presented in this dissertation can adopt any Sybil control mechanism.

[1]This is not to imply that every consensus protocol can be coupled with every Sybil control mechanism.

Nevertheless, it is a common misunderstanding that blockchain consensus protocols are based on these "Proof-of-X" mechanisms. It stems from the fact that Nakamoto consensus is achieved by chained PoWs, whereas PoW itself can simultaneously serve as a Sybil prevention mechanism. Those quorum-based protocols were then called "PoS protocols," because the projects using them for consensus also happen to adopt PoS for Sybil prevention. This has misled many people who are unfamiliar with consensus protocols and also confused the market since there are many projects who advertise their new "consensus protocols" that are solely built from some "Proof-of-X" mechanism.

2.6 Persistent State and Storage

Blockchain systems nowadays often use embedded key-value stores such as LevelDB or RocksDB for its underlying state persistence. There are two on-disk data structures that are often used for a general-purpose persistent key-value store: B^+-trees and Log-Structured Merge Trees (LSMs). B^+-trees have stable, predictable degradation as the data set grows large and supports fast read access, but they incur non-sequential small writes that sometimes get amplified in order to maintain balance. LSMs, on the other hand, are optimized for write-intensive workloads. They usually have high write-throughput as most of the writes are sequential and the resulting storage space is compact. An LSM does not have an explicit tree structure that organizes nodes as in B^+-trees. Instead, the merge tree serves as a conceptual hierarchy that directs how to merge-sort user data. The amount of merge-sorted data gets amplified as the level goes deeper and thus may cause severe write amplification during log compaction. Compared to B^+-trees, LSMs also have higher read amplification.

Both designs are intended for scenarios where the data set cannot fit in memory and the underlying secondary storage is much slower than memory. But servers and consumer devices have increasingly larger memories. Today, servers may have tens to thousands of gigabytes of memory with a secondary storage using flash or non-volatile memory technology. As a result, more applications can have all or at least most of their data set fully reside in memory, and this has ignited interest in exploring new data structures that better utilize the characteristics of the abundant memory [53, 158] or even non-volatile main memory [98, 109]. At the same time, solid state devices (SSDs) may also require changes in data structures on secondary storage to take advantage of the much faster access times [103, 140].

Some recent key-value store designs have eschewed on-disk indexes altogether, favoring a fast in-memory index and an unordered solution for persistence such as logging updates sequentially or slab allocation [47, 100, 108, 111]. While such designs can perform very well in the failure-free case, crash recovery involves reading large sections if not all of the disk, leading to lengthy recovery times. Checkpointing can improve recovery time but comes at a considerable overhead during normal operation (Section 6.3.4). As an example, FASTER [47] is a recent work that keeps an in-memory hash table and logs all writes to the disk. While the design utilizes the disk bandwidth well for intensive in-place updates, a user has to invoke checkpoints manually to make the store persistent, where both checkpointing and recovery take substantial time due to the lack of an on-disk index.

As a result, applications that can fit their data in memory but also need persistence currently have to choose between stores that maintain relatively slow

on-disk indexes and stores that have high failure recovery times. Therefore, later in this dissertation, we will also explore the design of key-value stores with the same persistence model as B^+-tree- or LSM-based approaches but whose performance is competitive with a log-based approach to persistence.

CHAPTER 3

HOTSTUFF: BFT CONSENSUS IN THE LENS OF BLOCKCHAIN

We revisited the classical approach where quorums of votes are formed to drive a common decision, repeated on individual slots to form a replicated log that tolerates Byzantine failures. By rethinking this approach from the new perspective of a blockchain (i.e., Nakamoto consensus), we consider if it is possible to simplify the practical protocol logic so that there are no explicit *view changes*, while safety can still be guaranteed with a unified handling of messages, as in a PoW-driven blockchain. Moreover, we want to preserve the property from quorum-based protocols that no PoW (Proof-of-Work) is requied in the agreement process.

Thus, we search for a "bridge" that connects classical consensus with Nakamoto consensus and hope that with new insights from both sides we can significantly reduce the complexity both in terms of the full protocol specification (engineering) and the algorithmic network cost (theory). This chapter will embark on this journey and show how we manage to achieve both with the HotStuff [154, 155] protocol, designed from scratch.

3.1 Model

We consider a system consisting of a fixed set of $n = 3f + 1$ *replicas*, indexed by $i \in [n]$ where $[n] = \{1, \ldots, n\}$. A set $F \subset [n]$ of up to $f = |F|$ replicas are Byzantine faulty, and the remaining ones are correct. We will often refer to the Byzantine

replicas as being coordinated by an *adversary*, which learns all internal state held by these replicas (including their cryptographic keys, see below).

Network communication is point-to-point, authenticated and reliable: one correct replica receives a message from another correct replica if and only if the latter sent that message to the former. When we refer to a "broadcast", it involves the broadcaster, if correct, sending the same point-to-point messages to all replicas, including itself. We adopt the *partial synchrony model* of Dwork et al. [69], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct replicas arrive within time Δ. Our protocol will ensure safety always, and will guarantee progress within a bounded duration after GST. (Guaranteeing progress before GST is impossible [76].) In practice, our protocol will guarantee progress if the system remains stable (i.e., if messages arrive within Δ time) for sufficiently long after GST, though assuming that it does so forever simplifies discussion.

Cryptographic primitives. HotStuff makes use of threshold signatures [31, 40, 141]. In a (k, n)-threshold signature scheme, there is a single public key held by all replicas, and each of the n replicas holds a distinct private key. The i-th replica can use its private key to contribute a *partial signature* $\rho_i \leftarrow tsign_i(m)$ on message m. Partial signatures $\{\rho_i\}_{i \in I}$, where $|I| = k$ and each $\rho_i \leftarrow tsign_i(m)$, can be used to produce a digital signature $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$ on m. Any other replica can verify the signature using the public key and the function *tverify*. We require that if $\rho_i \leftarrow tsign_i(m)$ for each $i \in I$, $|I| = k$, and if $\sigma \leftarrow tcombine(m, \{\rho_i\}_{i \in I})$, then $tverify(m, \sigma)$ returns true. However, given oracle access to oracles $\{tsign_i(\cdot)\}_{i \in [n] \setminus F}$, an adversary who queries $tsign_i(m)$ on strictly fewer than $k - f$ of these oracles has negligible probability of producing a signature σ for the message m (i.e.,

such that $tverify(m, \sigma)$ returns true). Throughout this chapter, we use a threshold of $k = 2f + 1$. Again, we will typically leave invocations of *tverify* implicit in our protocol descriptions.

We also require a cryptographic hash function h (also called a *message digest* function), which maps an arbitrary-length input to a fixed-length output. The hash function must be *collision resistant* [138], which informally requires that the probability of an adversary producing inputs m and m' such that $h(m) = h(m')$ is negligible. As such, $h(m)$ can serve as an identifier for a unique input m in the protocol.

Complexity measure. The complexity measure we care about is *authenticator complexity*, which specifically is the sum, over all replicas $i \in [n]$, of the number of authenticators received by replica i in the protocol to reach a consensus decision after GST. (Again, before GST, a consensus decision might not be reached at all in the worst case [76].) Here, an *authenticator* is either a partial signature or a signature. Authenticator complexity is a useful measure of communication complexity for several reasons. First, like bit complexity and unlike message complexity, it hides unnecessary details about the transmission topology. For example, n messages carrying one authenticator count the same as one message carrying n authenticators. Second, authenticator complexity is better suited than bit complexity for capturing costs in protocols like ours that reach consensus repeatedly, where each consensus decision (or each view proposed on the way to that consensus decision) is identified by a monotonically increasing counter. That is, because such a counter increases indefinitely, the bit complexity of a protocol that sends such a counter cannot be bounded. Third, since in practice, cryptographic operations to produce or verify digital signatures and to produce

or combine partial signatures are typically the most computationally intensive operations in protocols that use them, the authenticator complexity provides insight into the computational burden of the protocol, as well.

3.2 Basic HotStuff

HotStuff solves the State Machine Replication (SMR) problem. At the core of SMR is a protocol for deciding on a growing log of *command* requests by clients. A group of state-machine replicas apply commands in sequence order consistently. A client sends a command request to all replicas, and waits for responses from $(f + 1)$ of them. For the most part, we omit the client from the discussion, and defer to the standard literature for issues regarding numbering and de-duplication of client requests.

The Basic HotStuff solution is presented in Algorithm 2. The protocol works in a succession of *views* numbered with monotonically increasing view numbers. Each *viewNumber* has a unique dedicated leader known to all. Each replica stores a *tree* of pending commands as its local data structure. Each tree *node* contains a proposed command (or a batch of them), metadata associated with the protocol, and a *parent* link. The *branch* led by a given node is the path from the node all the way to the tree root by visiting parent links. During the protocol, a monotonically growing branch becomes *committed*. To become committed, the leader of a particular view proposing the branch must collect votes from a quorum of $(n - f)$ replicas in three phases, PREPARE, PRE-COMMIT, and COMMIT.

A key ingredient in the protocol is a collection of $(n - f)$ votes over a leader proposal, referred to as a *quorum certificate* (or "QC" in short). The QC is associ-

ated with a particular node and a view number. The *tcombine* utility employs a threshold signature scheme to generate a representation of $(n - f)$ signed votes as a single authenticator.

Below we give an operational description of the protocol logic by phases, followed by a precise specification in Algorithm 2, and conclude the section with safety, liveness, and complexity arguments.

3.2.1 Phases

PREPARE phase. The protocol for a new leader starts by collecting NEW-VIEW messages from $(n - f)$ replicas. The NEW-VIEW message is sent by a replica as it transitions into *viewNumber* (including the first view) and carries the highest *prepareQC* that the replica received ($\perp$ if none), as described below.

The leader processes these messages in order to select a branch that has the highest preceding view in which a *prepareQC* was formed. The leader selects the *prepareQC* with the highest view, denoted *highQC*, among the NEW-VIEW messages. Because *highQC* is the highest among $(n - f)$ replicas, no higher view could have reached a commit decision. The branch led by *highQC.node* is therefore safe.

Collecting NEW-VIEW messages to select a safe branch may be omitted by an incumbent leader, who may simply select its own highest *prepareQC* as *highQC*. We defer this optimization to Section 3.4 and only describe a single, unified leader protocol in this section. Note that, different from PBFT-like protocols, including this step in the leader protocol is straightforward, and it incurs the

same, linear overhead as all the other phases of the protocol, regardless of the situation.

The leader uses the CREATELEAF method to extend the tail of *highQC.node* with a new proposal. The method creates a new leaf node as a child and embeds a digest of the parent in the child node. The leader then sends the new node in a PREPARE message to all other replicas. The proposal carries *highQC* for safety justification.

Upon receiving the PREPARE message for the current view from the leader, replica r uses the SAFENODE predicate to determine whether to accept it. If it is accepted, the replica sends a PREPARE vote with a partial signature (produced by $tsign_r$) for the proposal to the leader.

SAFENODE predicate. The SAFENODE predicate is a core ingredient of the protocol. It examines a proposal message m carrying a QC justification $m.justify$, and determines whether $m.node$ is safe to accept. The safety rule to accept a proposal is the branch of $m.node$ extends from the currently locked node $lockedQC.node$. On the other hand, the liveness rule is the replica will accept m if $m.justify$ has a higher view than the current $lockedQC$. The predicate is true as long as either one of two rules holds.

PRE-COMMIT phase. When the leader receives $(n - f)$ PREPARE votes for the current proposal *curProposal*, it combines them into a *prepareQC*. The leader broadcasts *prepareQC* in PRE-COMMIT messages. A replica responds to the leader with PRE-COMMIT vote having a signed digest of the proposal.

COMMIT phase. The COMMIT phase is similar to PRE-COMMIT phase. When the leader receives $(n - f)$ PRE-COMMIT votes, it combines them into a *precommitQC* and broadcasts it in COMMIT messages; replicas respond to it with a COMMIT vote. Importantly, a replica becomes *locked* on the *precommitQC* at this point by setting its *lockedQC* to *precommitQC* (Line 25 of Algorithm 2). This is crucial to guard the safety of the proposal in case it becomes a consensus decision.

DECIDE phase. When the leader receives $(n - f)$ COMMIT votes, it combines them into a *commitQC*. Once the leader has assembled a *commitQC*, it sends it in a DECIDE message to all other replicas. Upon receiving a DECIDE message, a replica considers the proposal embodied in the *commitQC* a committed decision, and executes the commands in the committed branch. The replica increments *viewNumber* and starts the next view.

NEXTVIEW interrupt. In all phases, a replica waits for a message at view *viewNumber* for a timeout period, determined by an auxiliary NEXTVIEW(*viewNumber*) utility. If NEXTVIEW(*viewNumber*) interrupts waiting, the replica also increments *viewNumber* and starts the next view.

3.2.2 Data Structures

Messages. A message *m* in the protocol has a fixed set of fields that are populated using the MSG() utility shown in Algorithm 1. *m* is automatically stamped with *curView*, the sender's current view number. Each message has a type

$m.type \in$ {NEW-VIEW, PREPARE, PRE-COMMIT, COMMIT, DECIDE}. $m.node$ contains a proposed node (the leaf node of a proposed branch). There is an optional field $m.justify$. The leader always uses this field to carry the QC for the different phases. Replicas use it in NEW-VIEW messages to carry the highest *prepareQC*. Each message sent in a replica role contains a partial signature $m.partialSig$ by the sender over the tuple $\langle m.type, m.viewNumber, m.node\rangle$, which is added in the VOTEMSG() utility.

Quorum certificates. A Quorum Certificate (QC) over a tuple $\langle type, viewNumber, node\rangle$ is a data type that combines a collection of signatures for the same tuple signed by $(n - f)$ replicas. Given a QC qc, we use $qc.type$, $qc.viewNumber$, $qc.node$ to refer to the matching fields of the original tuple.

Tree and branches. Each command is wrapped in a node that additionally contains a parent link which could be a hash digest of the parent node. We omit the implementation details from the pseudocode. During the protocol, a replica delivers a message only after the branch led by the node is already in its local tree. In practice, a recipient who falls behind can catch up by fetching missing nodes from other replicas. For brevity, these details are also omitted from the pseudocode. Two branches are *conflicting* if neither one is an extension of the other. Two nodes are conflicting if the branches led by them are conflicting.

Bookkeeping variables. A replica uses additional local variables for bookkeeping the protocol state: (i) a *viewNumber*, initially 1 and incremented either by finishing a decision or by a NEXTVIEW interrupt; (ii) a locked quorum certificate *lockedQC*, initially $\perp$, storing the highest QC for which a replica voted COMMIT;

and (iii) a *prepareQC*, initially $\perp$, storing the highest QC for which a replica voted PRE-COMMIT. Additionally, in order to incrementally execute a committed log of commands, the replica maintains the highest node whose branch has been executed. This is omitted below for brevity.

3.2.3 Protocol Specification

The protocol given in Algorithm 2 is described as an iterated view-by-view loop. In each view, a replica performs phases in succession based on its role, described as a succession of "**as**" blocks. A replica can have more than one role. For example, a leader is also a (normal) replica. Execution of **as** blocks across roles can be proceeded concurrently. The execution of each **as** block is atomic. A NEXTVIEW interrupt aborts all operations in any **as** block, and jumps to the "Finally" block.

Algorithm 1 Utilities (for replica r).

```
1:  function MSG(type, node, qc)
2:      m.type ← type
3:      m.viewNumber ← curView
4:      m.node ← node
5:      m.justify ← qc
6:      return m
7:  function VOTEMSG(type, node, qc)
8:      m ← MSG(type, node, qc)
9:      m.partialSig ← tsign_r(⟨m.type, m.viewNumber, m.node⟩)
10:     return m
11: procedure CREATELEAF(parent, cmd)
12:     b.parent ← parent
13:     b.cmd ← cmd
14:     return b
15: function QC(V)
16:     qc.type ← m.type : m ∈ V
17:     qc.viewNumber ← m.viewNumber : m ∈ V
```

```
    qc.node ← m.node : m ∈ V
    qc.sig ← tcombine(⟨qc.type, qc.viewNumber, qc.node⟩, {m.partialSig | m ∈ V})
    return qc
function MATCHINGMSG(m, t, v)
    return (m.type = t) ∧ (m.viewNumber = v)
function MATCHINGQC(qc, t, v)
    return (qc.type = t) ∧ (qc.viewNumber = v)
function SAFENODE(node, qc)
    return (node extends from lockedQC.node) ∨   // safety rule
   (qc.viewNumber > lockedQC.viewNumber)   // liveness rule
```

Algorithm 2 Basic HotStuff protocol (for replica r).

```
for curView ← 1, 2, 3, . . . do
▷ PREPARE phase
    as a leader   // r = LEADER(curView)
    // we assume special NEW-VIEW messages from view 0
        wait for (n − f) NEW-VIEW messages:
            M ← {m | MATCHINGMSG(m, NEW-VIEW, curView − 1)}
        highQC ← ( arg max_{m∈M} {m.justify.viewNumber} ).justify
        curProposal ← CREATELEAF(highQC.node,
                                 client's command)
        broadcast MSG(PREPARE, curProposal, highQC)
    as a replica
        wait for message m from LEADER(curView)
            m : MATCHINGMSG(m, PREPARE, curView)
        if m.node extends from m.justify.node ∧
                    SAFENODE(m.node, m.justify) then
            send VOTEMSG(PREPARE, m.node, ⊥) to LEADER(curView)
▷ PRE-COMMIT phase
    as a leader
        wait for (n − f) votes:
            V ← {v | MATCHINGMSG(v, PREPARE, curView)}
        prepareQC ← QC(V)
        broadcast MSG(PRE-COMMIT, ⊥, prepareQC)
    as a replica
        wait for message m from LEADER(curView)
            m : MATCHINGQC(m.justify, PREPARE, curView)
        prepareQC ← m.justify
        send to LEADER(curView)
            VOTEMSG(PRE-COMMIT, m.justify.node, ⊥)
▷ COMMIT phase
```

19: **as** a leader
20: wait for $(n - f)$ votes:
$V \leftarrow \{v \mid \text{MATCHINGMSG}(v, \text{PRE-COMMIT}, curView)\}$
21: $precommitQC \leftarrow \text{QC}(V)$
22: broadcast $\text{MSG}(\text{COMMIT}, \perp, precommitQC)$
23: **as** a replica
24: wait for message m from $\text{LEADER}(curView)$
$m : \text{MATCHINGQC}(m.justify, \text{PRE-COMMIT}, curView)$
25: $lockedQC \leftarrow m.justify$
26: send to $\text{LEADER}(curView)$
$\text{VOTEMSG}(\text{COMMIT}, m.justify.node, \perp)$

▷ DECIDE phase

27: **as** a leader
28: wait for $(n - f)$ votes:
$V \leftarrow \{v \mid \text{MATCHINGMSG}(v, \text{COMMIT}, curView)\}$
29: $commitQC \leftarrow \text{QC}(V)$
30: broadcast $\text{MSG}(\text{DECIDE}, \perp, commitQC)$
31: **as** a replica
32: wait for message m from $\text{LEADER}(curView)$
$m : \text{MATCHINGQC}(m.justify, \text{COMMIT}, curView)$
33: execute new commands through $m.justify.node$,
respond to clients

▷ Finally

34: NEXTVIEW interrupt: goto this line if $\text{NEXTVIEW}(curView)$ is called during "wait for" in any phase
35: send $\text{MSG}(\text{NEW-VIEW}, \perp, prepareQC)$ to $\text{LEADER}(curView + 1)$

3.2.4 Safety, Liveness, and Complexity

Safety. We first define a quorum certificate qc to be *valid* if $tverify(\langle qc.type, qc.viewNumber, qc.node\rangle, qc.sig)$ is true.

Lemma 1. *For any valid qc_1, qc_2 in which $qc_1.type = qc_2.type$ and $qc_1.node$ conflicts with $qc_2.node$, we have $qc_1.viewNumber \neq qc_2.viewNumber$.*

Proof. To show a contradiction, suppose $qc_1.viewNumber = qc_2.viewNumber = v$. Because a valid QC can be formed only with $n - f = 2f + 1$ votes (i.e., partial signatures) for it, there must be a correct replica who voted twice in the same phase of v. This is impossible because the pseudocode allows voting only once for each phase in each view. □

Theorem 2. *If w and b are conflicting nodes, then they cannot be both committed, each by a correct replica.*

Proof. We prove this important theorem by contradiction. Let qc_1 denote a valid *commitQC* (i.e., $qc_1.type = \text{COMMIT}$) such that $qc_1.node = w$, and qc_2 denote a valid *commitQC* such that $qc_2.node = b$. Denote $v_1 = qc_1.viewNumber$ and $v_2 = qc_2.viewNumber$. By Lemma 1, $v_1 \neq v_2$. W.l.o.g. assume $v_1 < v_2$.

We will now denote by v_s the lowest view higher than v_1 for which there is a valid *prepareQC*, qc_s (i.e., $qc_s.type = \text{PREPARE}$) where $qc_s.viewNumber = v_s$, and $qc_s.node$ conflicts with w. Formally, we define the following predicate for any *prepareQC*:

$$E(prepareQC) :=(v_1 < prepareQC.viewNumber \leq v_2) \\ \wedge \ (prepareQC.node \text{ conflicts with } w).$$

We can now set the *first* switching point qc_s:

$$qc_s := \arg\min_{prepareQC} \{prepareQC.viewNumber \mid prepareQC \text{ is valid} \wedge E(prepareQC)\}.$$

Note that, by assumption such a qc_s must exist; for example, qc_s could be the *prepareQC* formed in view v_2.

Of the correct replicas that sent a partial result $tsign_r(\langle qc_1.type, qc_1.viewNumber, qc_1.node\rangle)$, let r be the first that contributed $tsign_r(\langle qc_s.type, qc_s.viewNumber,$

$qc_s.node\rangle$); such an r must exist since otherwise, one of $qc_1.sig$ and $qc_s.sig$ could not have been created. During view v_1, replica r updates its lock *lockedQC* to a *precommitQC* on w at Line 25 of Algorithm 2. Due to the minimality of v_s, the lock that replica r has on the branch led by w is not changed before qc_s is formed. Otherwise r must have seen some other *prepareQC* with lower view because Line 17 comes before Line 25, contradicting to the minimality. Now consider the invocation of SAFENODE in the PREPARE phase of view v_s by replica r, with a message m carrying $m.node = qc_s.node$. By assumption, $m.node$ conflicts with $lockedQC.node$, and so the disjunct at Line 26 of Algorithm 1 is false. Moreover, $m.justify.viewNumber > v_1$ would violate the minimality of v_s, and so the disjunct in Line 27 of Algorithm 1 is also false. Thus, SAFENODE must return false and r cannot cast a PREPARE vote on the conflicting branch in view v_s, a contradiction. □

Liveness. There are two functions left undefined in the previous section: LEADER and NEXTVIEW. Their definition will *not* affect safety of the protocol, though they do matter to liveness. Before giving candidate definitions for them, we first show that after GST, there is a bounded duration T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached. Below, we say that qc_1 and qc_2 *match* if qc_1 and qc_2 are valid, $qc_1.node = qc_2.node$, and $qc_1.viewNumber = qc_2.viewNumber$.

Lemma 3. *If a correct replica is locked such that lockedQC = precommitQC, then at least $f + 1$ correct replicas voted for some prepareQC matching lockedQC.*

Proof. Suppose replica r is locked on *precommitQC*. Then, $(n - f)$ votes were cast for the matching *prepareQC* in the PREPARE phase (Line 10 of Algorithm 2), out of which at least $f + 1$ were from correct replicas. □

Theorem 4. *After GST, there exists a bounded time period T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached.*

Proof. Starting in a new view, the leader collects $(n - f)$ NEW-VIEW messages and calculates its *highQC* before broadcasting a PREPARE messsage. Suppose among all replicas (including the leader itself), the highest kept lock is *lockedQC* = *precommitQC**. By Lemma 3, we know there are at least $f + 1$ correct replicas that voted for a *prepareQC** matching *precommitQC**, and have already sent them to the leader in their NEW-VIEW messages. Thus, the leader must learn a matching *prepareQC** in at least one of these NEW-VIEW messages and use it as *highQC* in its PREPARE message. By the assumption, all correct replicas are synchronized in their view and the leader is non-faulty. Therefore, all correct replicas will vote in the PREPARE phase, since in SAFENODE, the condition on Line 27 of Algorithm 1 is satisfied (even if the *node* in the message conflicts with a replica's stale *lockedQC.node*, and so Line 26 is not). Then, after the leader assembles a valid *prepareQC* for this view, all replicas will vote in all the following phases, leading to a new decision. After GST, the duration T_f for these phases to complete is of bounded length.

The protocol is Optimistically Responsive because there is no explicit "wait-for-Δ" step, and the logical disjunction in SAFENODE is used to override a stale lock with the help of the three-phase paradigm. □

We now provide simple constructions for LEADER and NEXTVIEW that suffice to ensure that after GST, eventually a view will be reached in which the leader is correct and all correct replicas remain in this view for T_f time. It suffices for LEADER to return some deterministic mapping from view number to a replica, eventually rotating through all replicas. A possible solution for NEXTVIEW is to utilize an exponential back-off mechanism that maintains a timeout interval. Then a timer is set upon entering each view. When the timer goes off without making any decision, the replica doubles the interval and calls NEXTVIEW to advance the view. Since the interval is doubled at each time, the waiting intervals of all correct replicas will eventually have at least T_f overlap in common, during which the leader could drive a decision.

Livelessness with two-phases. We now briefly demonstrate an infinite non-deciding scenario for a "two-phase" HotStuff. This explains the necessity for introducing a synchronous delay in Casper and Tendermint, and hence for abandoning (Optimistic) Responsiveness.

In the two-phase HotStuff variant, we omit the PRE-COMMIT phase and proceed directly to COMMIT. A replica becomes locked when it votes on a *prepareQC*. Suppose, in view v, a leader proposes b. It completes the PREPARE phase, and some replica r_v votes for the *prepareQC*, say qc, such that $qc.node = b$. Hence, r_v becomes locked on qc. An asynchronous network scheduling causes the rest of the replicas to move to view $v + 1$ without receiving qc.

We now repeat ad infinitum the following single-view transcript. We start view $v + 1$ with only r_v holding the highest *prepareQC* (i.e. qc) in the system. The new leader l collects new-view messages from $2f + 1$ replicas excluding

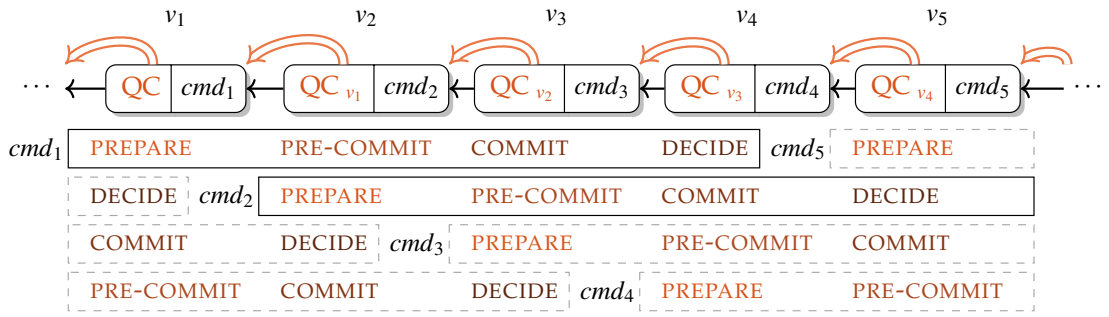


Figure 3.1: Chained HotStuff is a pipelined Basic HotStuff where a QC can serve in different phases simultaneously.

r_v. The highest *prepareQC* among these, qc', has view $v-1$ and $b' = qc'.node$ conflicts with b. l then proposes b'' which extends b', to which $2f$ honest replicas respond with a vote, but r_v rejects it because it is locked on qc, b'' conflicts with b and qc' is lower than qc. Eventaully, $2f$ replicas give up and move to the next view. Just then, a faulty replica responds to l's proposal, l then puts together a $prepareQC(v+1, b'')$ and one replica, say r_{v+1} votes for it and becomes locked on it.

Complexity. In each phase of HotStuff, only the leader broadcasts to all replicas while the replicas respond to the sender once with a partial signature to certify the vote. In the leader's message, the QC consists of a proof of $(n-f)$ votes collected previously, which can be encoded by a single threshold signature. In a replica's response, the partial signature from that replica is the only authenticator. Therefore, in each phase, there are $O(n)$ authenticators received in total. As there is a constant number of phases, the overall complexity per view is $O(n)$.

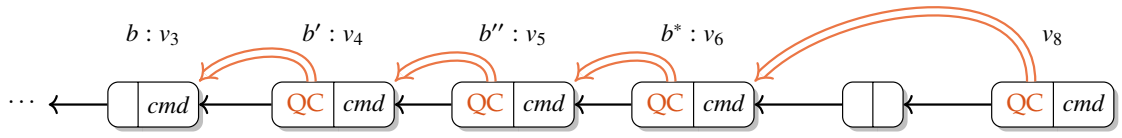


Figure 3.2: The nodes at views v_4, v_5, v_6 form a Three-Chain. The node at view v_8 does not make a valid One-Chain in Chained HotStuff (but it is a valid One-Chain after relaxation in the algorithm of Section 3.4).

3.3 Chained HotStuff

It takes three phases for a Basic HotStuff leader to commit a proposal. These phases are not doing "useful" work except collecting votes from replicas, and they are all very similar. In Chained HotStuff, we improve the Basic HotStuff protocol utility while at the same time considerably simplifying it. The idea is to change the view on *every* prepare *phase*, so each proposal has its own view. This reduces the number of message types and allows for pipelining of decisions. A similar approach for message type reduction was suggested in Casper [2].

More specifically, in Chained HotStuff the votes over a PREPARE phase are collected in a view by the leader into a *genericQC*. Then the *genericQC* is relayed to the leader of the next view, essentially delegating responsibility for the next phase, which would have been PRE-COMMIT, to the next leader. However, the next leader does not actually carry a PRE-COMMIT phase, but instead initiates a new PREPARE phase and adds its own proposal. This PREPARE phase for view $v + 1$ simultaneously serves as the PRE-COMMIT phase for view v. The PREPARE phase for view $v + 2$ simultaneously serves as the PRE-COMMIT phase for view $v + 1$ and as the COMMIT phase for view v. This is possible because all the phases have identical structure.

The pipeline of Basic HotStuff protocol phases embedded in a chain of Chained HotStuff proposals is depicted in Figure 3.1. Views v_1, v_2, v_3 of Chained HotStuff serve as the PREPARE, PRE-COMMIT, and COMMIT Basic HotStuff phases for cmd_1 proposed in v_1. This command becomes committed by the end of v_4. Views v_2, v_3, v_4 serve as the three Basic HotStuff phases for cmd_2 proposed in v_2, and it becomes committed by the end of v_5. Additional proposals generated in these phases continue the pipeline similarly, and are denoted by dashed boxes. In Figure 3.1, a single arrow denotes the *b.parent* field for a node b, and a double arrow denotes *b.justify.node*.

Hence, there are only two types of messages in Chained HotStuff, a NEW-VIEW message and generic-phase GENERIC message. The GENERIC QC functions in all logically pipelined phases. We next explain the mechanisms in the pipeline to take care of locking and committing, which occur only in the COMMIT and DECIDE phases of Basic HotStuff.

Dummy nodes. The *genericQC* used by a leader in some view *viewNumber* may not directly reference the proposal of the preceding view (*viewNumber* − 1). The reason is that the leader of a preceding view fails to obtain a QC, either because there are conflicting proposals, or due to a benign crash. To simplify the tree structure, CREATELEAF extends *genericQC.node* with blank nodes up to the height (the number of parent links on a node's branch) of the proposing view, so view-numbers are equated with node heights. As a result, the QC embedded in a node b may not refer to its parent, i.e., *b.justify.node* may not equal *b.parent* (the last node in Figure 3.2).

One-Chain, Two-Chain, and Three-Chain. When a node b^* carries a QC that refers to a direct parent, i.e., $b^*.justify.node = b^*.parent$, we say that it forms a *One-Chain*. Denote by $b'' = b^*.justify.node$. Node b^* forms a *Two-Chain*, if in addition to forming a One-Chain, $b''.justify.node = b''.parent$. It forms a *Three-Chain*, if b'' forms a Two-Chain.

Looking at chain $b = b'.justify.node$, $b' = b''.justify.node$, $b'' = b^*.justify.node$, ancestry gaps might occur at any one of the nodes. These situations are similar to a leader of Basic HotStuff failing to complete any one of three phases, and getting interrupted to the next view by NEXTVIEW.

If b^* forms a One-Chain, the PREPARE phase of b'' has succeeded. Hence, when a replica votes for b^*, it should remember $genericQC \leftarrow b^*.justify$. We remark that it is safe to update *genericQC* even when a One-Chain is not direct, so long as it is higher than the current *genericQC*. In the implementation code described in Section 3.4, we indeed update *genericQC* in this case.

If b^* forms a Two-Chain, then the PRE-COMMIT phase of b' has succeeded. The replica should therefore update $lockedQC \leftarrow b''.justify$. Again, we remark that the lock can be updated even when a Two-Chain is not direct—safety will not break—and indeed, this is given in the implementation code in Section 3.4.

Finally, if b^* forms a Three-Chain, the COMMIT phase of b has succeeded, and b becomes a committed decision.

Algorithm 3 shows the pseudocode for Chained HotStuff. The proof of safety given by Theorem 5 in Appendix A is similar to the one for Basic HotStuff. We require the QC in a valid node refers to its ancestor. For brevity, we assume the constraint always holds and omit checking in the code.

Algorithm 3 Chained HotStuff protocol.

procedure CREATELEAF($parent, cmd, qc$)
  $b.parent \leftarrow$ branch extending with blanks from $parent$ to height $curView$
  $b.cmd \leftarrow cmd$
  $b.justify \leftarrow qc$
  return b

for $curView \leftarrow 1, 2, 3, \ldots$ **do**
▷ GENERIC phase
  as a leader // r = LEADER($curView$)
  // M is the set of messages collected at the end of previous view by the leader of this view
    $highQC \leftarrow \left(\arg\max_{m \in M}\{m.justify.viewNumber\}\right).justify$
    if $highQC.viewNumber > genericQC.viewNumber$ **then**
      $genericQC \leftarrow highQC$
    $curProposal \leftarrow$ CREATELEAF($genericQC.node$, client's command, $genericQC$)
    // PREPARE phase
    broadcast MSG(GENERIC, $curProposal, \perp$)
  as a replica
    wait for message m from LEADER($curView$)
      m : MATCHINGMSG(m, GENERIC, $curView$)
    $b^* \leftarrow m.node$; $b'' \leftarrow b^*.justify.node$;
    $b' \leftarrow b''.justify.node$; $b \leftarrow b'.justify.node$
    if SAFENODE($b^*, b^*.justify$) **then**
      send VOTEMSG(GENERIC, $b^*, \perp$) to LEADER($curView + 1$)
    // start PRE-COMMIT phase on b^*'s parent
    if $b^*.parent = b''$ **then**
      $genericQC \leftarrow b^*.justify$
    // start COMMIT phase on b^*'s grandparent
    if $(b^*.parent = b'') \wedge (b''.parent = b')$ **then**
      $lockedQC \leftarrow b''.justify$
    // start DECIDE phase on b^*'s great-grandparent
    if $(b^*.parent = b'') \wedge (b''.parent = b') \wedge (b'.parent = b)$ **then**
      execute new commands through b, respond to clients
  as the next leader
    wait for all messages:
      $M \leftarrow \{m \mid$ MATCHINGMSG(m, GENERIC, $curView$)$\}$
      until there are $(n - f)$ votes:
      $V \leftarrow \{v \mid v.partialSig \neq \perp \wedge v \in M\}$
    $genericQC \leftarrow$ QC(V)

▷ Finally
27: NEXTVIEW interrupt: goto this line if NEXTVIEW(*curView*) is called during "wait for" in any phase
28: send MSG(GENERIC, ⊥, *genericQC*) to LEADER(*curView* + 1)

3.4 Implementation

HotStuff is a practical protocol for building efficient SMR systems. Because of its simplicity, we can easily turn Algorithm 3 into an event-driven-style specification that is almost like the code skeleton for a prototype implementation.

As shown in Algorithm 4, the code is further simplified and generalized by extracting the liveness mechanism from the body into a module named *Pacemaker*. Instead of the next leader always waiting for a *genericQC* at the end of the GENERIC phase before starting its reign, this logic is delegated to the Pacemaker. A stable leader can skip this step and streamline proposals across multiple heights. Additionally, we relax the direct parent constraint for maintaining the highest *genericQC* and *lockedQC*, while still preserving the requirement that the QC in a valid node always refers to its ancestor. The proof of correctness is similar to Chained HotStuff and we also defer it to Appendix B.

Data structures. Each replica u keeps track of the following main state variables:

$V[\cdot]$	mapping from a node to its votes.
$vheight$	height of last voted node.
b_{lock}	locked node (similar to *lockedQC*).
b_{exec}	last executed node.
qc_{high}	highest known QC (similar to *genericQC*) kept by a Pacemaker.
b_{leaf}	leaf node kept by a Pacemaker.

It also keeps a constant b_0, the same genesis node known by all correct replicas. To bootstrap, b_0 contains a hard-coded QC for itself, $b_{lock}, b_{exec}, b_{leaf}$ are all initialized to b_0, and qc_{high} contains the QC for b_0.

Pacemaker. A Pacemaker is a mechanism that guarantees progress after GST. It achieves this through two ingredients.

The first one is "synchronization", bringing all correct replicas, and a unique leader, into a common height for a sufficiently long period. The usual synchronization mechanism in the literature [35, 43, 69] is for replicas to increase the count of Δ's they spend at larger heights, until progress is being made. A common way to deterministically elect a leader is to use a rotating leader scheme in which all correct replicas keep a predefined leader schedule and rotate to the next one when the leader is demoted.

Second, a Pacemaker needs to provide the leader with a way to choose a proposal that will be supported by correct replicas. As shown in Algorithm 5, after a view change, in ONRECEIVENEWVIEW, the new leader collects NEW-VIEW messages sent by replicas through ONNEXTSYNCVIEW to discover the highest QC to satisfy the second part of the condition in ONRECEIVEPROPOSAL for live-

ness (Line 18 of Algorithm 4). During the same view, however, the incumbent leader will chain the new node to the end of the leaf last proposed by itself, where no NEW-VIEW message is needed. Based on some application-specific heuristics (to wait until the previously proposed node gets a QC, for example), the current leader invokes ONBEAT to propose a new node carrying the command to be executed.

It is worth noting that even if a bad Pacemaker invokes ONPROPOSE arbitrarily, or selects a parent and a QC capriciously, and against any scheduling delays, safety is always guaranteed. Therefore, safety guaranteed by Algorithm 4 alone is entirely decoupled from liveness by any potential instantiation of Algorithm 5.

Algorithm 4 Event-driven HotStuff (for replica u).

1: **procedure** CREATELEAF($parent, cmd, qc, height$)
2: $b.parent \leftarrow parent$; $b.cmd \leftarrow cmd$;
3: $b.justify \leftarrow qc$; $b.height \leftarrow height$; **return** b
4: **procedure** UPDATE(b^*)
5: $b'' \leftarrow b^*.justify.node$; $b' \leftarrow b''.justify.node$
6: $b \leftarrow b'.justify.node$
// PRE-COMMIT phase on b''
7: UPDATEQCHIGH($b^*.justify$)
8: **if** $b'.height > b_{lock}.height$ **then**
9: $b_{lock} \leftarrow b'$ // COMMIT phase on b'
10: **if** $(b''.parent = b') \wedge (b'.parent = b)$ **then**
11: ONCOMMIT(b)
12: $b_{exec} \leftarrow b$ // DECIDE phase on b
13: **procedure** ONCOMMIT(b)
14: **if** $b_{exec}.height < b.height$ **then**
15: ONCOMMIT($b.parent$); EXECUTE($b.cmd$)
16: **procedure** ONRECEIVEPROPOSAL($\text{MSG}_v(\text{GENERIC}, b_{new}, \perp)$)
17: **if** $b_{new}.height > vheight \wedge (b_{new}$ extends $b_{lock} \vee$
18: $b_{new}.justify.node.height > b_{lock}.height)$ **then**
19: $vheight \leftarrow b_{new}.height$
20: SEND(GETLEADER(), $\text{VOTEMSG}_u(\text{GENERIC}, b_{new}, \perp)$)
21: UPDATE(b_{new})

22: **procedure** ONRECEIVEVOTE(m = VOTEMSG$_v$(GENERIC, b, $\perp$))
23: **if** $\exists \langle v, \sigma' \rangle \in V[b]$ **then return** // avoid duplicates
24: $V[b] \leftarrow V[b] \cup \{\langle v, m.partialSig \rangle\}$ // collect votes
25: **if** $|V[b]| \geq n - f$ **then**
26: $qc \leftarrow$ QC($\{\sigma \mid \langle v', \sigma \rangle \in V[b]\}$)
27: UPDATEQCHIGH(qc)
28: **function** ONPROPOSE(b_{leaf}, cmd, qc_{high})
29: $b_{new} \leftarrow$ CREATELEAF($b_{leaf}, cmd, qc_{high}, b_{leaf}.height + 1$)
// send to all replicas, including u itself
30: BROADCAST(MSG$_u$(GENERIC, $b_{new}, \perp$))
31: **return** b_{new}

Algorithm 5 Code skeleton for a Pacemaker (for replica u).

// We assume Pacemaker in all correct replicas will have synchronized leadership after GST.
1: **function** GETLEADER // ...specified by the application
2: **procedure** UPDATEQCHIGH(qc'_{high})
3: **if** $qc'_{high}.node.height > qc_{high}.node.height$ **then**
4: $qc_{high} \leftarrow qc'_{high}$
5: $b_{leaf} \leftarrow qc_{high}.node$
6: **procedure** ONBEAT(cmd)
7: **if** u = GETLEADER() **then**
8: $b_{leaf} \leftarrow$ ONPROPOSE(b_{leaf}, cmd, qc_{high})
9: **procedure** ONNEXTSYNCVIEW
10: send MSG(NEW-VIEW, $\perp, qc_{high}$) to GETLEADER()
11: **procedure** ONRECEIVENEWVIEW(MSG(NEW-VIEW, $\perp, qc'_{high}$))
12: UPDATEQCHIGH(qc'_{high})

Algorithm 6 UPDATE replacement for two-phase HotStuff.

1: **procedure** UPDATE(b^*)
2: $b' \leftarrow b^*.justify.node$; $b \leftarrow b'.justify.node$
3: UPDATEQCHIGH($b^*.justify$)
4: **if** $b'.height > b_{lock}.height$ **then** $b_{lock} \leftarrow b'$
5: **if** $(b'.parent = b)$ **then** ONCOMMIT(b); $b_{exec} \leftarrow b$

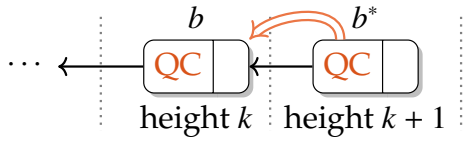
Two-phase HotStuff variant. To further demonstrate the flexibility of the HotStuff framework, Algorithm 6 shows the two-phase variant of HotStuff. Only the UPDATE procedure is affected, a Two-Chain is required for reaching a commit decision, and a One-Chain determines the lock. As discussed above (Section 3.2.4), this two-phase variant loses Optimistic Responsiveness, and is similar to Tendermint/Casper. The benefit is fewer phases, while liveness may be addressed by incorporating in Pacemaker a wait based on maximum network delay. See Section 3.5.3 for further discussion.

3.5 One-Chain and Two-Chain BFT Protocols

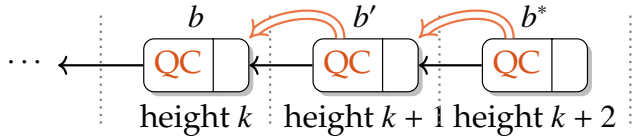
In this section, we examine four BFT replication protocols spanning four decades of research in Byzantine fault tolernace, casting them into a chained framework similar to Chained HotStuff.

Figure 3.3 provides a birds-eye view of the commit rules of five protocols we consider, including HotStuff.

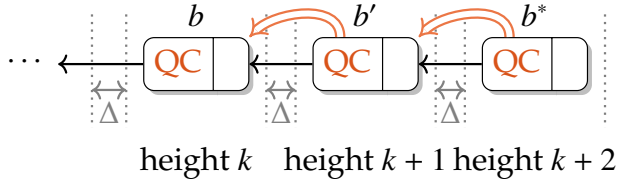
In a nutshell, the commit rule in DLS [69] is One-Chain, allowing a node to be committed only by its own leader. The commit rules in PBFT [43], Tendermint [35, 36] and Casper [38] are almost identical, and consist of Two-Chains. They differ in the mechanisms they introduce for liveness, PBFT has leader "proofs" of quadratic size (no Linearity), Tendermint and Casper introduce a mandatory Δ delay before each leader proposal (no Optimistic Responsiveness). HotStuff uses a Three-Chain rule, and has a linear leader protocol without delay.



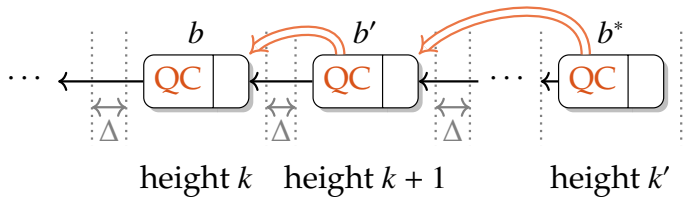
(a) One-Chain (DLS, 1988)



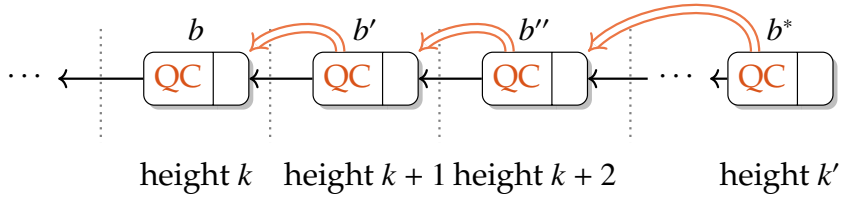
(b) Two-Chain (PBFT, 1999)



(c) Two-Chain w/ delay (Tendermint, 2016)



(d) Two-Chain w/ delay (Casper, 2017)



(e) Three-Chain (HotStuff, 2018)

Figure 3.3: Commit rules for different BFT protocols.

3.5.1 DLS

The simplest commit rule is a One-Chain. Modeled after Dwork, Lynch, and Stockmeyer (DLS), the first known asynchronous Byzantine Consensus solution, this rule is depicted in Figure 3.3(a). A replica becomes locked in DLS on the highest node it voted for.

Unfortunately, this rule would easily lead to a deadlock if at some height, a leader equivocates, and two correct replicas became locked on the conflicting proposals at that height. Relinquishing either lock is unsafe unless there are $2f + 1$ that indicate they did not vote for the locked value.

Indeed, in DLS only the leader of each height can itself reach a commit decision by the One-Chain commit rule. Thus, only the leader itself is harmed if it has equivocated. Replicas can relinquish a lock either if $2f + 1$ replicas did not vote for it, or if there are conflicting proposals (signed by the leader). The unlocking protocol occurring at the end of each height in DLS turns out to be fairly complex and expensive. Together with the fact that only the leader for a height can decide, in the best scenario where no fault occurs and the network is timely, DLS requires n leader rotations, and $O(n^4)$ message transmissions, per single decision. While it broke new ground in demonstrating a safe asynchronous protocol, DLS was not designed as a practical solution.

3.5.2 PBFT

Modeled after PBFT, a more practical appraoch uses a Two-Chain commit rule, see Figure 3.3(b). When a replica votes for a node that forms a One-Chain, it

becomes locked on it. Conflicting One-Chains at the same height are simply not possible, as each has a QC, hence the deadlock situation of DLS is avoided.

However, if one replica holds a higher lock than others, a leader may not know about it even if it collects information from $n - f$ replicas. This could prevent leaders from reaching decisions ad infinitum, purely due to scheduling. To get "unstuck", the PBFT unlocks all replicas by carrying a *proof* consisting of the highest One-Chain's by $2f + 1$ replicas. This proof is quite involved, as explained below.

The original PBFT, which has been open-sourced [43] and adopted in several follow up works [26,92], a leader proof contains a set of messages collected from $n-f$ replicas reporting the highest One-Chain each member voted for. Each One-Chain contains a QC, hence the total communication cost is $O(n^3)$. Harnessing signature combining methods from [40,133], SBFT [80] reduces this cost to $O(n^2)$ by turning each QC to a single value.

In the PBFT variant in [44], a leader proof contains the highest One-Chain the leader collected from the quorum only once. It also includes one signed value from each member of the quorum, proving that it did not vote for a higher One-Chain. Broadcasting this proof incurs communication complexity $O(n^2)$. Note that whereas the signatures on a QC may be combined into a single value, the proof as a whole cannot be reduced to constant size because messages from different members of the quorum may have different values.

In both variants, a correct replica unlocks even it has a higher One-Chain than the leader's proof. Thus, a correct leader can force its proposal to be ac-

cepted during period of synchrony, and liveness is guaranteed. The cost is quadratic communication per leader replacement.

3.5.3 Tendermint and Casper

Tendermint has a Two-Chain commit rule identical to PBFT, and Casper has a Two-Chain rule in which the leaf does not need to have a QC to direct parent. That is, in Casper, Figure 3.3(c,d) depicts the commit rules for Tendermint and Casper, respectively.

In both methods, a leader simply sends the highest One-Chain it knows along with its proposal. A replica unlocks a One-Chain if it receives from the leader a higher one.

However, because correct replicas may not vote for a leader's node, to guarantee progress a new leader must obtain the highest One-Chain by waiting the maximal network delay. Otherwise, if leaders only wait for the first $n - f$ messages to start a new height, there is no progress guarantee. Leader delays are inherent both in Tendermint and in Casper, in order to provide liveness.

This simple leader protocol embodies a linear leap in the communication complexity of the leader protocol, which HotStuff borrows from. As already mentioned above, a QC could be captured in a single value using threshold-signatures, hence a leader can collect and disseminate the highest One-Chain with linear communication complexity. However, crucially, due to the extra QC step, HotStuff does not require the leader to wait the maximal network delay.

3.6 Evaluation

We have implemented HotStuff as a library in roughly 4K lines of C++ code. Most noticeably, the core consensus logic specified in the pseudocode consumes only around 200 lines. In this section, we will first examine baseline throughput and latency by comparing to a state-of-art system, BFT-SMaRt [26]. We then focus on the message cost for view changes to see our advantages in this scenario.

3.6.1 Setup

We conducted our experiments on Amazon EC2 using `c5.4xlarge` instances. Each instance had 16 vCPUs supported by Intel Xeon Platinum 8000 processors. All cores sustained a Turbo CPU clock speed up to 3.4GHz. We ran each replica on a single VM instance, and so BFT-SMaRt, which makes heavy use of threads, was allowed to utilize 16 cores per replica, as in their original evaluation [26]. The maximum TCP bandwidth measured by `iperf` was around 1.2 Gigabytes per second. We did not throttle the bandwidth in any run. The network latency between two machines was less than 1 ms.

Our prototype implementation of HotStuff uses secp256k1 for all digital signatures in both votes and quorum certificates. BFT-SMaRt uses hmac-sha1 for MACs (Message Authentication Codes) in the messages during normal operation and uses digital signatures in addition to MACs during a view change.

All results for HotStuff reflect end-to-end measurement from the clients. For BFT-SMaRt, we used the micro-benchmark programs `ThroughputLatency-Server` and `ThroughputLatencyClient` from the BFT-SMaRt website

(`https://github.com/bft-smart/library`). The client program measures end-to-end latency but not throughput, while the server-side program measures both throughput and latency. We used the throughput results from servers and the latency results from clients.

3.6.2 Base Performance

We first measured throughput and latency in a setting commonly seen in the evaluation of other BFT replication systems. We ran 4 replicas in a configuration that tolerates a single failure, i.e., $f = 1$, while varying the operation request rate until the system saturated. This benchmark used empty (zero-sized) operation requests and responses and triggered no view changes; we expand to other settings below. Although our responsive HotStuff is three-phase, we also run its two-phase variant as an additional baseline, because the BFT-SMaRt baseline has only two phases.

Figure 3.4 depicts three batch sizes for both systems, 100, 400, and 800, though because these systems have different batching schemes, these numbers mean slightly different things for each system. BFT-SMaRt drives a separate consensus decision for each operation, and batches the messages from multiple consensus protocols. Therefore, it has a typical L-shaped latency/throughput performance curve. HotStuff batches multiple operations in each node, and in this way, mitigates the cost of digital signatures per decision. However, above 400 operations per batch, the latency incurred by batching becomes higher than the cost of the crypto. Despite these differences, both three-phase ("`HS3-`") and two-phase ("`HS2-`") HotStuff achieves comparable latency performance to BFT-

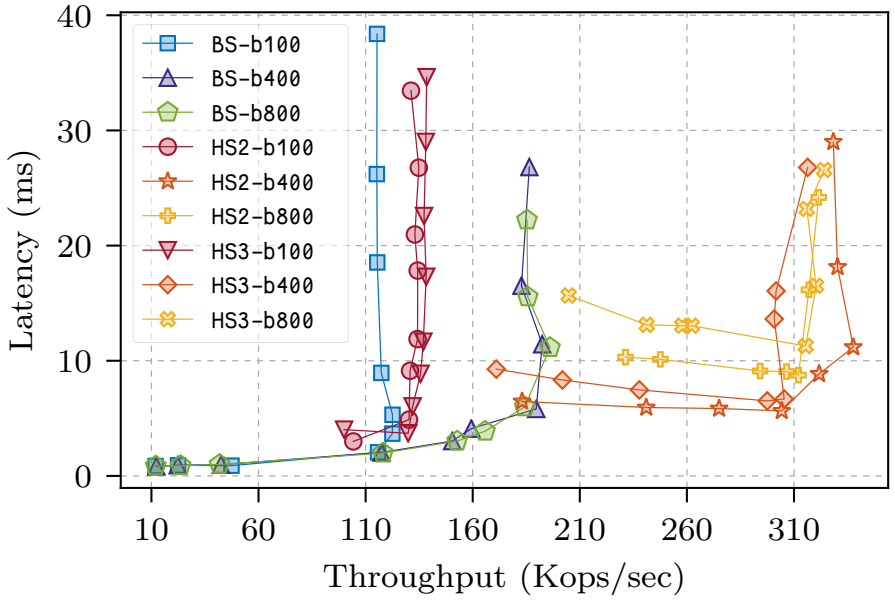


Figure 3.4: Throughput vs. latency with different choices of batch size, 4 replicas, 0/0 payload.

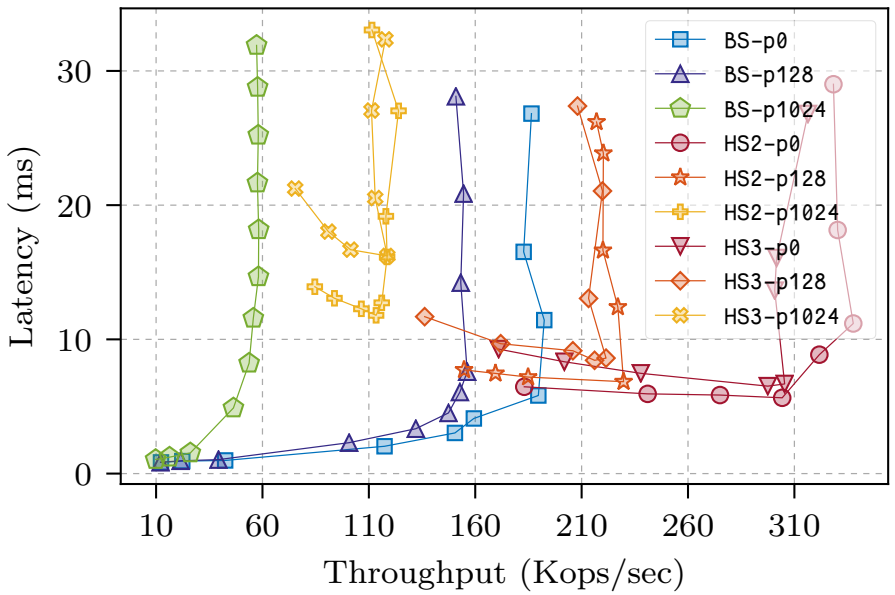


Figure 3.5: Throughput vs. latency with different choices of payload size, 4 replicas, batch size of 400.

SMaRt ("BS-") for all three batch sizes, while their maximum throughput noticeably outperformed BFT-SMaRt.

For batch sizes of 100 and 400, the lowest-latency HotStuff point provides latency and throughput that are better than the latency and throughput simultaneously achievable by BFT-SMaRT at its highest throughput, while incurring a small increase in latency. This increase is partly due to the batching strategy employed by HotStuff: It needs three additional full batches (two in the two-phase variant) to arrive at a decision on a batch. Our experiments kept the number of outstanding requests high, but the higher the batch size, the longer it takes to fill the batching pipeline. Practical deployments could be further optimized to adapt the batch size to the number of outstanding operations.

Figure 3.5 depicts three client request/reply payload sizes (in bytes) of 0/0, 128/128, and 1024/1024, denoted "p0", "p128", and "p1024" respectively. At all payload sizes, both three-phase and two-phase HotStuff outperformed BFT-SMaRt in throughput, with similar or comparable latency.

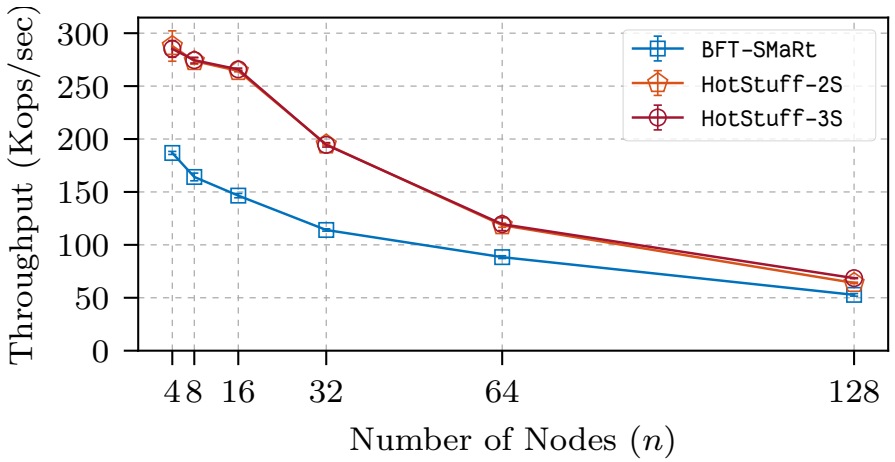
Notice BFT-SMaRt uses MACs based on symmetric crypto that is orders of magnitude faster than the asymmetric crypto in digital signatures used by HotStuff, and also three-phase HotStuff has more round trips compared to two-phase PBFT variant used by BFT-SMaRt. Yet HotStuff is still able to achieve comparable latency and much higher throughput. Below we evaluate both systems in more challenging situations, where the performance advantages of HotStuff will become more pronounced.

3.6.3 Scalability

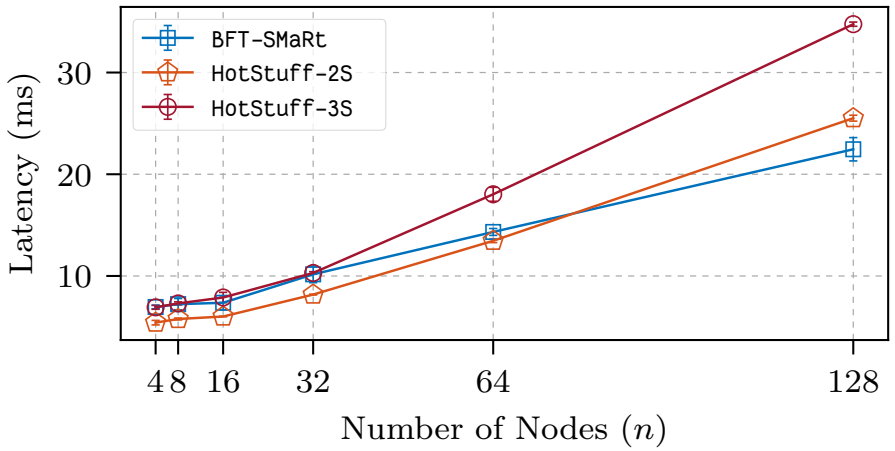
To evaluate the scalability of HotStuff in various dimensions, we performed three experiments. For the baseline, we used zero-size request/response payloads while varying the number of replicas. The second evaluation repeated the baseline experiment with 128-byte and 1024-byte request/response payloads. The third test repeated the baseline (with empty payloads) while introducing network delays between replicas that were uniformly distributed in 5ms ± 0.5ms or in 10ms ± 1.0ms, implemented using `NetEm` (see `https://www.linux.org/docs/man8/tc-netem.html`). For each data point, we repeated five runs with the same setting and show error bars to indicate the standard deviation for all runs.

The first setting is depicted in Figure 3.6a (throughput) and Figure 3.6b (latency). Both three-phase and two-phase HotStuff show consistently better throughput than BFT-SMaRt, while their latencies are still comparable to BFT-SMaRt with graceful degradation. The performance scales better than BFT-SMaRt when $n < 32$. This is because we currently still use a list of secp256k1 signatures for a QC. In the future, we plan to reduce the cryptographic computation overhead in HotStuff by using a fast threshold signature scheme.

The second setting with payload size 128 or 1024 bytes is denoted by "`p128`" or "`p1024`" in Figure 3.7a (throughput) and Figure 3.7b (latency). Due to its quadratic bandwidth cost, the throughput of BFT-SMaRt scales worse than HotStuff for reasonably large (1024-byte) payload size.



(a) Throughput



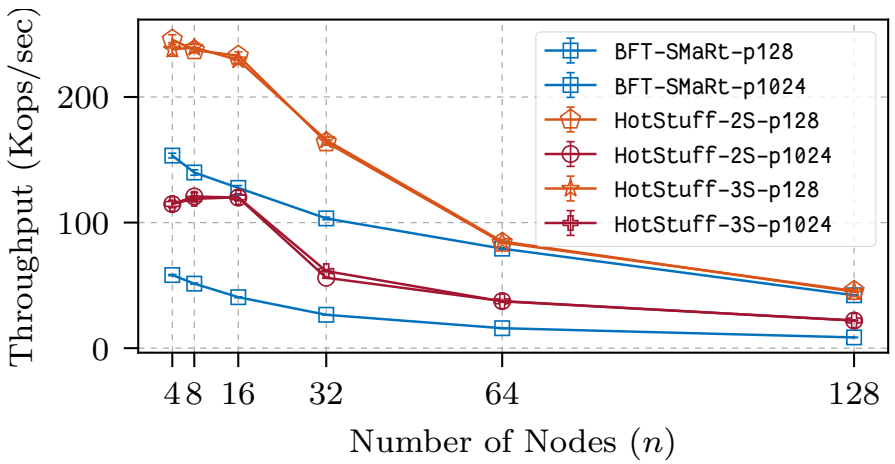
(b) Latency

Figure 3.6: Scalability with 0/0 payload, batch size of 400.

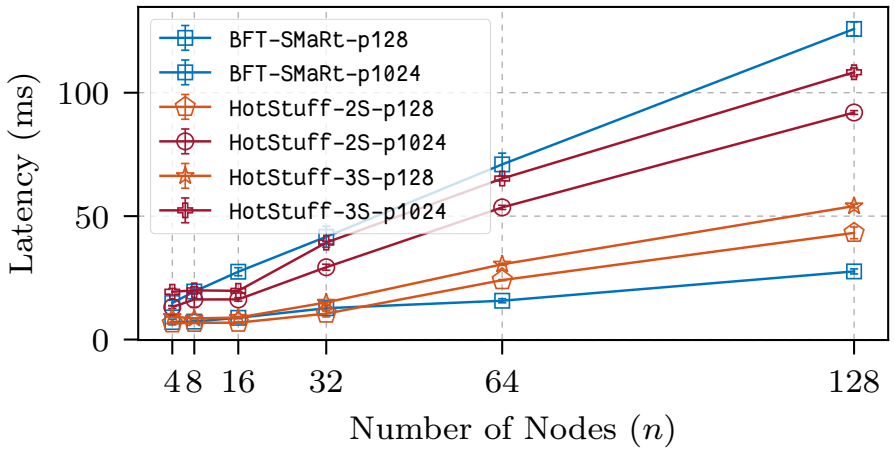
The third setting is shown in Figure 3.8a (throughput) and Figure 3.8b (latency) as "`5ms`" or "`10ms`". Again, due to the larger use of communication in BFT-SMaRt, HotStuff consistently outperformed BFT-SMaRt in both cases.

3.6.4 View Change

To evaluate the communication complexity of leader replacement, we counted the number of *MAC or signature verifications* performed within BFT-SMaRt's



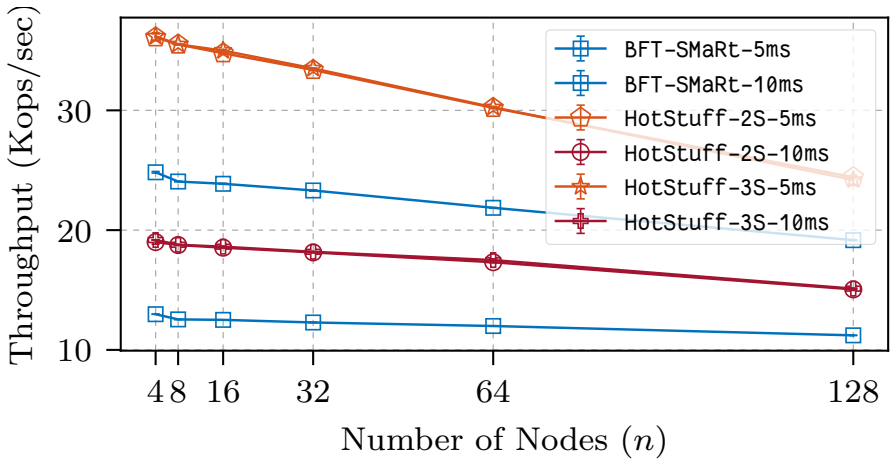
(a) Throughput



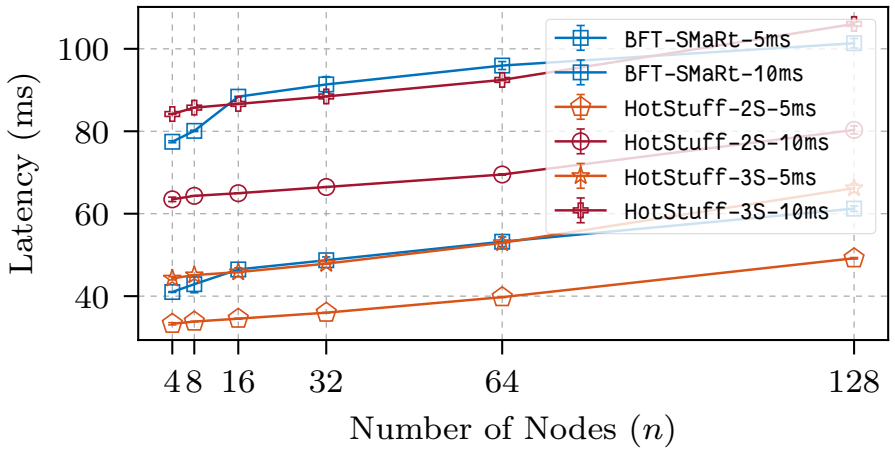
(b) Latency

Figure 3.7: Scalability for 128/128 payload or 1024/1024 payload, with batch size of 400.

view-change protocol. Our evaluation strategy was as follows. We injected a view change into BFT-SMaRt every one thousand decisions. We instrumented the BFT-SMaRt source code to count the number of verifications upon receiving and processing messages within the view-change protocol. Beyond communication complexity, this measurement underscores the cryptographic computation load associated with transferring these authenticated values.



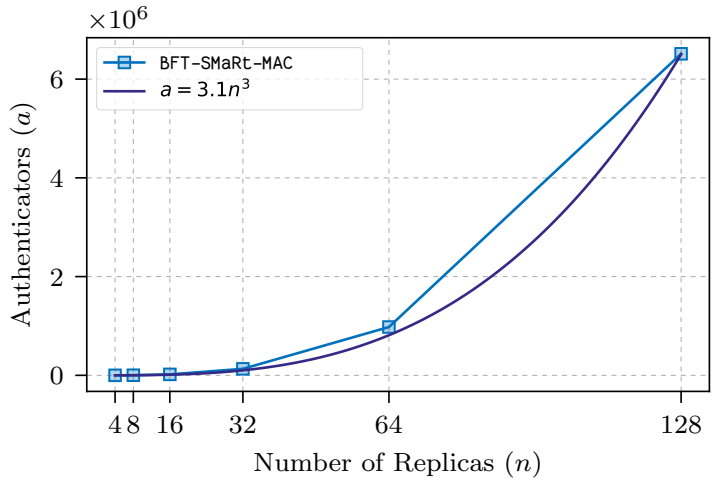
(a) Throughput



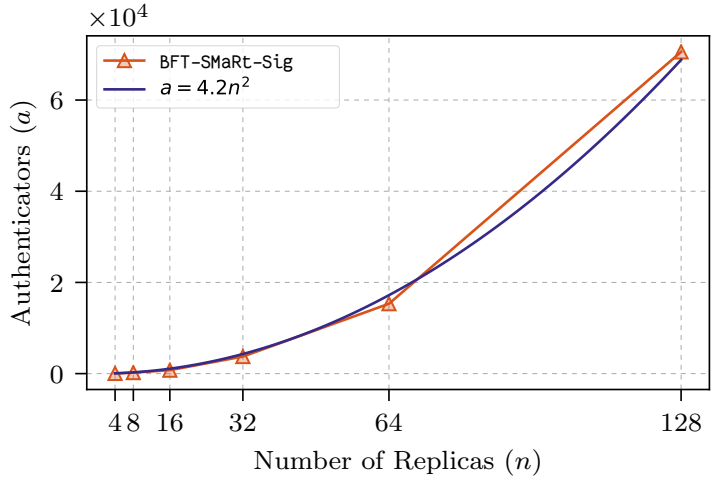
(b) Latency

Figure 3.8: Scalability for inter-replica latency 5ms ± 0.5ms or 10ms ± 1.0ms, with 0/0 payload, batch size of 400.

Figure 3.9a and Figure 3.9b show the number of extra authenticators (MACs and signatures, respectively) processed for each view change, where "extra" is defined to be those authenticators that would not be sent if the leader remained stable. Note that HotStuff has no "extra" authenticators by this definition, since the number of authenticators remains the same regardless of whether the leader stays the same or not. The two figures show that BFT-SMaRt uses cubic numbers of MACs and quadratic numbers of signatures. HotStuff does not require extra authenticators for view changes and so is omitted from the graph.



(a) MACs



(b) Signatures

Figure 3.10: Number of extra authenticators used for each BFT-SMaRt view change.

Evaluating the real-time performance of leader replacement is tricky. First, BFT-SMaRt got stuck when triggering frequent view changes; our authenticator-counting benchmark had to average over as many successful view changes as possible before the system got stuck, repeating the experiment many times. Second, the actual elapsed time for leader replacement depends highly on timeout parameters and the leader-election mechanism. It is therefore impossible to provide a meaningful comparison.

3.7 Related work

Reaching consensus in face of Byzantine failures was formulated as the Byzantine Generals Problem by Lamport et al. [97], who also coined the term "Byzantine failures". The first synchronous solution was given by Pease et al. [126], and later improved by Dolev and Strong [65]. The improved protocol has $O(n^3)$ communication complexity, which was shown optimal by Dolev and Reischuk [64]. A leader-based synchronous protocol that uses randomness was given by Katz and Koo [87], showing an expected constant-round solution with $(n-1)/2$ resilience.

Meanwhile, in the asynchronous settings, Fischer et al. [76] showed that the problem is unsolvable deterministically in asynchronous setting in face of a single failure. Furthermore, an $(n-1)/3$ resilience bound for any asynchronous solution was proven by Ben-Or [24]. Two approaches were devised to circumvent the impossibility. One relies on randomness, initially shown by Ben-Or [24], using independently random coin flips by processes until they happen to converge to consensus. Later works used cryptographic methods to share an unpredictable coin and drive complexities down to constant expected rounds, and $O(n^3)$ communication [40].

The second approach relies on partial synchrony, first shown by Dwork, Lynch, and Stockmeyer (DLS) [69]. This protocol preserves safety during asynchronous periods, and after the system becomes synchronous, DLS guarantees termination. Once synchrony is maintained, DLS incurs $O(n^4)$ total communication and $O(n)$ rounds per decision.

State machine replication relies on consensus at its core to order client requests so that correct replicas execute them in this order. The recurring need for consensus in SMR led Lamport to devise Paxos [95], a protocol that operates an efficient pipeline in which a stable leader drives decisions with linear communication and one round-trip. A similar emphasis led Castro and Liskov [43, 44] to develop an efficient leader-based Byzantine SMR protocol named PBFT, whose stable leader requires $O(n^2)$ communication and two round-trips per decision, and the leader replacement protocol incurs $O(n^3)$ communication. PBFT has been deployed in several systems, including BFT-SMaRt [26]. Kotla et al. introduced an optimistic linear path into PBFT in a protocol named Zyzzyva [92], which was utilized in several systems, e.g., Upright [49] and Byzcoin [91]. The optimistic path has linear complexity, while the leader replacement protocol remains $O(n^3)$. Abraham et al. [6] later exposed a safety violation in Zyzzyva, and presented fixes [7, 80]. On the other hand, to also reduce the complexity of the protocol itself, Song et al. proposed Bosco [146], a simple one-step protocol with low latency on the optimistic path, requiring $5f + 1$ replicas. SBFT [80] introduces an $O(n^2)$ communication view-change protocol that supports a stable leader protocol with optimistically linear, one round-trip decisions. It reduces the communication complexity by harnessing two methods: a collector-based communication paradigm by Reiter [133], and signature combining via threshold cryptography on protocol votes by Cachin et al. [40].

A leader-based Byzantine SMR protocol that employs randomization was presented by Ramasamy et al. [132], and a leaderless variant named HoneyBadgerBFT was developed by Miller et al. [115]. At their core, these randomized Byzantine solutions employ randomized asynchronous Byzantine consensus, whose best known communication complexity was $O(n^3)$ (see above), amortiz-

ing the cost via batching. However, most recently, based on the idea of HotStuff, a parallel submission to PODC'19 [11] further improves the communication complexity to $O(n^2)$.

Bitcoin's core is a protocol known as Nakamoto Consensus [119], a synchronous protocol with only probabilistic safety guarantee and no finality (see analysis in [8, 78, 123]). It operates in a *permissionless* model where participants are unknown, and resilience is kept via Proof-of-Work. As described above, recent blockchain solutions hybridize Proof-of-Work solutions with classical BFT solutions in various ways [9, 38, 73, 79, 82, 91, 125]. The need to address rotating leaders in these hybrid solutions and others provide the motivation behind HotStuff.

CHAPTER 4

SNOW: SCALABLE AND PROBABILISTIC LEADERLESS BFT CONSENSUS THROUGH METASTABILITY

Aside from the similaries we discovered in the prior chapter, there are some fundamental differences between classical and Nakamoto consensus, one of which is finality versus safety. In a blockchain, a user can achieve finality by waiting for a number of blocks that follow the block with the submitted transaction. The well-known *six-block rule* ensures the termination of consensus but only guarantees safety probabilistically. While the safety is guarded, the actual probability is not as good as one would imagine, even given only a small fraction of network adversaries (Figure 4.2). Another difference is that blockchain nodes do not need to exchange state information by direct messaging for the agreement, but do PoW locally and disseminate new blocks by *gossip*.

Given these unique traits of Nakamoto consensus, we consider if we can design a protocol that also trades off deterministic safety for a more loosely formed agreement, such that it can achieve a much stronger safety than Nakamoto consensus, while at the same time benefit from the scalability in network communication. It behaves like a stochastic version of the quorum-based protocol, without explicit intersections in voting. We think this could be another new paradigm for consensus protocols, that achieves agreement by random network sampling. In this chapter, we will present the Snow protocol family [135, 136] with these interesting potential advantages.

4.1 Model and Goals

Key Guarantees

Safety Unlike classical consensus protocols, and similar to longest-chain-based protocols such as Nakamoto [119], we adopt an ε-safety guarantee that is probabilistic. In practice, this probabilistic guarantee is as strong as traditional safety guarantees, since appropriately small choices of ε can render consensus failure negligible, lower than the probability of hardware failure due to random events. Figure 4.1 shows how the portion (f/n) of misbehaving participants (or computation power) affects the probability of system safety failure (decision of two conflicting proposals), given a choice of finality.

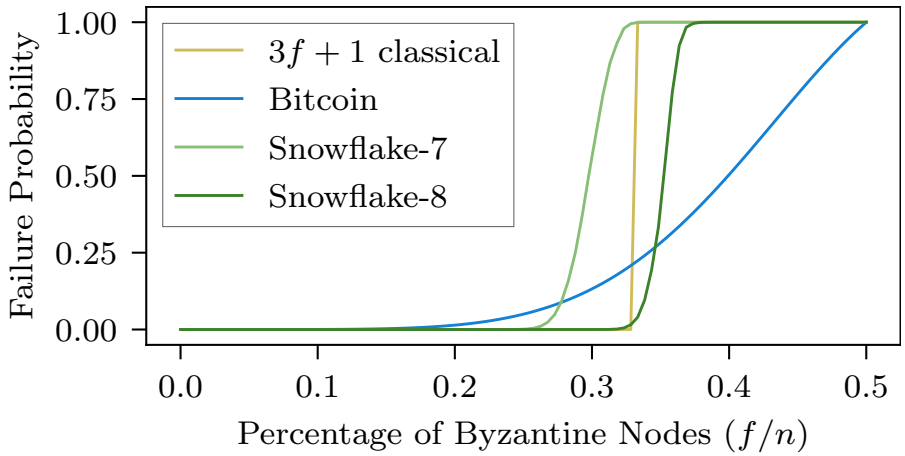


Figure 4.1: Classical BFT protocols that tolerate f failures will encounter total safety failure when the threshold is exceeded even by one additional node. The Bitcoin curve shows a typical finality choice for Bitcoin where a block is considered final when it is "buried" in a branch having 6 additional blocks compared to any other competing forks. Snowflake belongs to the Snow family, and it is configured with $k = 10, \beta = 150$. Snowflake-7,8 uses $\alpha = 7$ and $\alpha = 8$ respectively (see Section 4.2 for the definition of k, α and β).

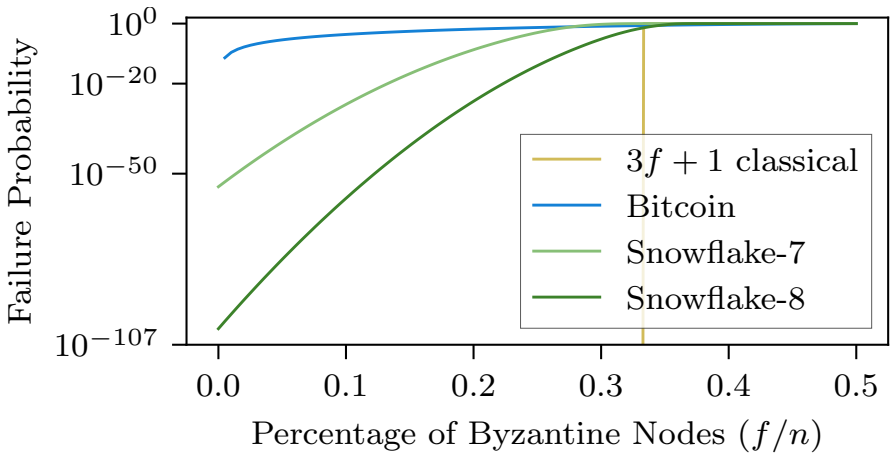


Figure 4.2: Figure 4.1 with log-scaled y-axis.

Liveness All our protocols provide a non-zero probability guarantee of termination within a bounded amount of time. This bounded guarantee is similar to various protocols such as Ben-Or [24] and longest-chain protocols. In particular, for Nakamoto consensus, the number of required blocks for a transaction increases exponentially with the number of adversarial nodes, with an asymptote at $f = n/2$ wherein the number is infinite. In other words, the time required for finality approaches ∞ as f approaches $n/2$ (Figure 4.3). Furthermore, the required number of rounds is calculable ahead of time, as to allow the system designer to tune liveness at the expense of safety. Lastly, unlike traditional consensus protocols and similar to Nakamoto, our protocols benefit from lower adversarial presence, as discussed in property P3 below.

Formal Guarantees: Let the system be parameterized for an ε safety failure probability under a maximum expected f number of adversarial nodes. Let $O(\log n) < t_{max} < \infty$ be the upper bound of the execution of the protocols. The Snow protocols then provide the following guarantees:

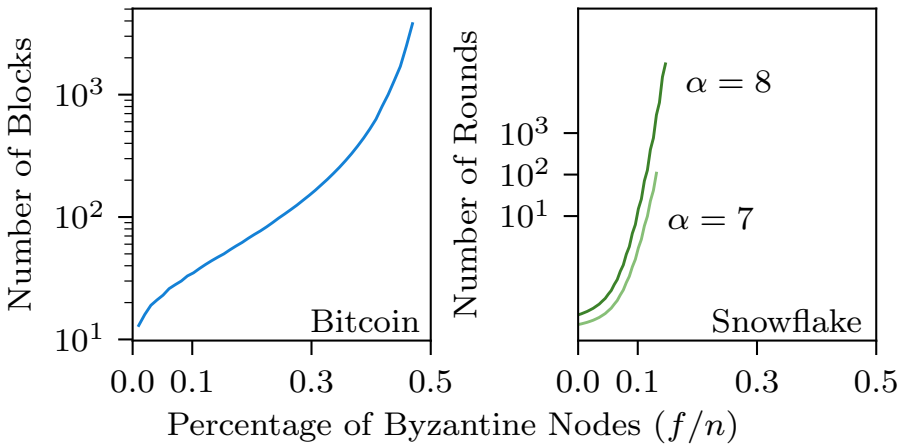


Figure 4.3: The relation between f/n and the convergence speed, given $\varepsilon = 10^{-20}$. The left figure shows the expected number of blocks to guarantee ε in Bitcoin, which, counter to commonly accepted folk wisdom, is not a constant 6, but depends on adversary size to withhold the same ε. The right figure shows the maximum number of rounds required by Snowflake, where being different from Bitcoin, the asymptote is below 0.5 and varies by the choice of parameters.

- **P1. Safety.** When decisions are made by any two correct nodes, they decide on conflicting transactions with negligible probability ($\leq \varepsilon$).
- **P2. Liveness (Upper Bound).** Snow protocols terminate with a strictly positive probability within t_{max} rounds.
- **P3. Liveness (Strong).** If $f \leq O(\sqrt{n})$, then Snow protocols terminate with high probability ($\geq 1 - \varepsilon$) in $O(\log n)$ rounds.

Network In the standard definition of asynchrony [24], message transmission time is finite, but the distribution is unspecified (and thus the delivery time can be unbounded for some messages). This implies that the scheduling of message transmission itself could behave arbitrarily, and potentially even maliciously (with full asynchrony). We use a modified version of this model, which is well-accepted [22, 68, 77, 88, 102] in the analysis of epidemic networks and gossip-based stochastic systems. In particular, we fix the distribution of message delay

to that of the exponential distribution. We note that, just like in the standard asynchronous model, there is a strictly non-zero probability that any correct node may execute its next local round only after an arbitrarily large amount of time has passed. Furthermore, we also note that scheduling only applies to correct nodes, and the adversary may execute arbitrarily, as discussed later.

Achieving Liveness Classical consensus that works with asynchrony does not get stuck in a single phase of voting because the vote initiator always polls votes from all known participants and waits for $n - f$ responses. In our system, however, nodes operate via subsampling, hence it is possible for a single sample to select a majority of adversarial nodes, and therefore the node gets stuck waiting for the responses. To ensure liveness, a node should be able to wait with some timeout. Therefore, our protocols are synchronous in order to guarantee liveness. Lastly, note that Nakamoto consensus is synchronous, in which the required difficulty of PoW is dependent on the maximum network delay [123].

Adversary The adversarial nodes execute under their own internal scheduler, which is unbounded in speed, meaning that all adversarial nodes can execute at any infinitesimally small point in time, unlike correct nodes. The adversary can view the state of every honest node at all times and can instantly modify the state of all adversarial nodes. It cannot, however, schedule or modify communication between correct nodes. Finally, we make zero assumptions about the behavior of the adversary, meaning that it can choose any execution strategy of its liking. In short, the adversary is computationally bounded (it cannot forge digital signatures) but otherwise is point-to-point informationally unbounded (knows all state) and round-adaptive (can modify its strategy at any time).

Sybil Attacks Consensus protocols provide their guarantees based on assumptions that only a fraction of participants are adversarial. These bounds could be violated if the network is naively left open to arbitrary participants. In particular, a Sybil attack [67], wherein a large number of identities are generated by an adversary, could be used to exceed the bounds.

A long line of work, including PBFT [43], treats the Sybil problem separately from consensus, and rightfully so, as Sybil control mechanisms are distinct from the underlying, more complex agreement protocol[1]. In fact, to our knowledge, only Nakamoto-style consensus has "baked-in" Sybil prevention as part of its consensus, made possible by PoW [17], which requires miners to continuously stake a hardware investment. Other protocols, discussed in Section 5.7, rely on PoS or PoA. The consensus protocols presented in this chapter can adopt any Sybil control mechanism. The current deployment uses an established PoS based mechanism [79]. To participate, one has to stake in funds, while the entrance is finalized as the result of consensus on the staking chain (another subsystem based on Snowball). The platform uses Annual Percentage Yield style reward to incentivize participants and also handles bootstrapping, minting, and inflation control. Details are beyond the scope of this chapter, whose focus is on a novel paradigm of the core consensus algorithm.

Flooding Attacks Flooding/spam attacks are a problem for any distributed system. Without a protection mechanism, an attacker can generate large numbers of transactions and flood protocol data structures, consuming storage. There are a multitude of techniques to deter such attacks, including network-

[1]This is not to imply that every consensus protocol can be coupled/decoupled with every Sybil control mechanism.

layer protection, PoA, local PoW and economic mechanisms. In Avalanche, we use transaction fees, making such attacks costly even if the attacker is sending money back to addresses under its control.

Additional Assumptions We do not assume that all members are known to all participants, but rather may temporarily have some discrepancies in network view. We quantify the bounds on the discrepancy in [136]. We assume a safe bootstrapping mechanism, similar to that of Bitcoin, that enables a node to connect with sufficiently many correct nodes to acquire a statistically unbiased view of the network. We do not assume a PKI. We make standard cryptographic assumptions related to digital signatures and hash functions.

4.2 Protocol Design

We start with a non-BFT protocol called Slush and progressively build up to Snowflake and Snowball, all based on the same common majority-based metastable voting mechanism. These protocols are single-decree consensus protocols of increasing robustness. We provide full specifications for the protocols in this section, and defer the analysis and formal proofs to [136].

4.2.1 Slush: Introducing Metastability

The core of our approach is a single-decree consensus protocol, inspired by gossip protocols. The simplest protocol, Slush, is the foundation of this family, shown in Figure 4.4. Slush is *not* tolerant to Byzantine faults, only crash-faults

(CFT), but serves as an illustration for the BFT protocols that follow. For ease of exposition, we will describe the operation of Slush using a decision between two conflicting colors, red and blue.

In Slush, a node starts out initially in an uncolored state. Upon receiving a transaction from a client, an uncolored node updates its own color to the one carried in the transaction and initiates a query. To perform a query, a node picks a small, constant sized (k) sample of the network uniformly at random, and sends a query message. The constancy of k may sound counter-intuitive. Our proof [136] shows that if we fix k and grow the number of nodes in network, the safety probability reaches an asymptote that is under ε (resulting from tail-bounds that contain k) and thus never exceeds it. Upon receiving a query, an uncolored node adopts the color in the query, responds with that color, and initiates its own query, whereas a colored node simply responds with its current color. Once the querying node collects k responses, it checks if a fraction $\geq \alpha$ are for the same color, where $\alpha > \lfloor k/2 \rfloor$ is a protocol parameter. If the α threshold is met and the sampled color differs from the node's own color, the node flips to that color. It then goes back to the query step, and initiates a subsequent round of query, for a total of m rounds. Finally, the node decides the color it ended up with at time m.

Slush has a few properties of interest. First, it is almost *memoryless*: a node retains no state between rounds other than its current color, and in particular maintains no history of interactions with other peers. Second, unlike traditional consensus protocols that query every participant, every round involves sampling just a small, constant-sized slice of the network at random. Third, Slush makes progress under any network configuration (even fully bivalent state, i.e.

```
1:  procedure ONQUERY(v, col′)
2:      if col = ⊥ then col ← col′
3:      RESPOND(v, col)
4:  procedure SLUSHLOOP(u, col_0 ∈ {R, B, ⊥})
5:      col ← col_0 // initialize with a color
6:      for r ∈ {1 . . . m} do
7:          // if ⊥, skip until ONQUERY sets the color
8:          if col = ⊥ then continue
9:          // randomly sample from the known nodes
10:         K ← SAMPLE(N\u, k)
11:         P ← [QUERY(v, col)  for v ∈ K]
12:         for col′ ∈ {R, B} do
13:             if P.COUNT(col′) ≥ α then
14:                 col ← col′
15:     ACCEPT(col)
```

Figure 4.4: Slush protocol. Timeouts elided for readability.

```
1:  procedure SNOWFLAKELOOP(u, col_0 ∈ {R, B, ⊥})
2:      col ← col_0, cnt ← 0
3:      while undecided do
4:          if col = ⊥ then continue
5:          K ← SAMPLE(N\u, k)
6:          P ← [QUERY(v, col)  for v ∈ K]
7:          maj ← false
8:          for col′ ∈ {R, B} do
9:              if P.COUNT(col′) ≥ α then
10:                 maj ← true
11:                 if col′ ≠ col then col ← col′, cnt ← 1
12:                 else cnt++
13:                 if cnt ≥ β then ACCEPT(col′)
14:         if maj = false then cnt ← 0
```

Figure 4.5: Snowflake.

50/50 split between colors), since random perturbations in sampling will cause one color to gain a slight edge and repeated samplings afterwards will build upon and amplify that imbalance. Finally, if m is chosen high enough, Slush ensures that all nodes will be colored identically with high probability (w.h.p.). Each node has a constant, predictable communication overhead per round, and m grows logarithmically with n.

The Slush protocol does not provide a strong safety guarantee in the presence of Byzantine nodes. In particular, if the correct nodes develop a preference for one color, a Byzantine adversary can attempt to flip nodes to the opposite so as to keep the network in balance, preventing a decision. We address this in our first BFT protocol that introduces more state storage at the nodes.

4.2.2 Snowflake: BFT

Snowflake augments Slush with a single counter that captures the strength of a node's conviction in its current color. This per-node counter stores how many consecutive samples of the network by that node have all yielded the same color. A node accepts the current color when its counter reaches β, another security parameter. Figure 4.5 shows the amended protocol, which includes the following modifications:

1. Each node maintains a counter *cnt*;
2. Upon every color change, the node resets *cnt* to 0;
3. Upon every successful query that yields $\geq \alpha$ responses for the same color as the node, the node increments *cnt*.

When the protocol is correctly parameterized for a given threshold of Byzantine nodes and a desired ε-guarantee, it can ensure both safety (P1) and liveness (P2, P3). As we later show, there exists an irreversible state after which a decision is inevitable. Correct nodes begin to commit past the irreversible state to adopt the same color, w.h.p. For additional intuition, which we do not expand

```
1:  procedure SNOWBALLLOOP(u, col_0 ∈ {R, B, ⊥})
2:      col ← col_0, lastcol ← col_0, cnt ← 0
3:      d[R] ← 0, d[B] ← 0
4:      while undecided do
5:          if col = ⊥ then continue
6:          K ← SAMPLE(N\u, k)
7:          P ← [QUERY(v, col)   for v ∈ K]
8:          maj ← false
9:          for col' ∈ {R, B} do
10:             if P.COUNT(col') ≥ α then
11:                 maj ← true
12:                 d[col']++
13:                 if d[col'] > d[col] then col ← col'
14:                 if col' ≠ lastcol then lastcol ← col', cnt ← 1
15:                 else cnt++
16:                 if cnt ≥ β then ACCEPT(col')
17:         if maj = false then cnt ← 0
```

Figure 4.6: Snowball.

in this chapter, there also exists a phase-shift point, where the Byzantine nodes lose ability to keep network in a bivalent state.

4.2.3 Snowball: Adding Confidence

Snowflake's notion of state is ephemeral: the counter gets reset with every color flip. Snowball augments Snowflake with *confidence counters* that capture the number of queries that have yielded a threshold result for their corresponding color (Figure 4.6). A node decides if it gets β consecutive chits for a color. However, it only changes preference based on the total accrued confidence. The differences between Snowflake and Snowball are as follows:

1. Upon every successful query, the node increments its confidence counter for that color.

2. A node switches colors when the confidence counter of its current color becomes lower than the confidence counter of the new color.

Multi-Value Consensus Our binary consensus protocol could support multi-value consensus by running logarithmic binary instances, one for each bit of the proposed value. However, such theoretical reduction might not be efficient in practice. Instead, we could directly incorporate multi-values as multi-colors in the protocol, where safety analysis could still be generalized.

In the appendices of this work [136]'s full paper, we sketch a leaderless initialization mechanism, which in expectation uses $O(\log n)$ rounds under the assumption that the network is synchronized, and also discuss churn and view discrepancies. While the design of initialization mechanisms is interesting, note that it is not necessary for a decentralized payment system, as we show in Section 5.

CHAPTER 5

AVALANCHE: INTERNET-SCALE PEER-TO-PEER PAYMENT SYSTEM

Using Snowball consensus, we have implemented a payment system, Avalanche [135, 136], which supports Bitcoin transactions. In this chapter, we describe the design and sketch how the implementation can support the value transfer primitive at the center of cryptocurrencies.

Nodes send messages to one another directly, as relaying messages is prone to attack. Having each node maintain $n - 1$ network connections might seem problematic, but our protocol does not saturate all connections. At any given moment, only k connections are active, leaving most of them idle.

In a cryptocurrency setting, cryptographic signatures enforce that only a key owner is able to create a transaction that spends a particular coin. Since correct clients follow the protocol as prescribed and never double spend coins, in Avalanche, they are guaranteed both safety and liveness for their *virtuous* transactions. In contrast, liveness is not guaranteed for *rogue* transactions, submitted by Byzantine clients, which conflict with one another. Such decisions may stall in the network, but have no safety impact on virtuous transactions. We show that this is a sensible tradeoff, and that the resulting system is sufficient for building complex payment systems.

5.1 Avalanche: Adding a DAG

Avalanche consists of multiple single-decree Snowball instances instantiated as a multi-decree protocol that maintains a dynamic, append-only directed acyclic graph (DAG) of all known transactions. The DAG has a single sink that is the

genesis vertex. Maintaining a DAG provides two significant benefits. First, it improves efficiency, because a single vote on a DAG vertex implicitly votes for all transactions on the path to the genesis vertex. Second, it also improves security, because the DAG intertwines the fate of transactions, similar to the Bitcoin blockchain. This renders past decisions difficult to undo without the approval of correct nodes.

When a client creates a transaction, it names one or more *parents*, which are included inseparably in the transaction and form the edges of the DAG. The parent-child relationships encoded in the DAG may, but do not need to, correspond to application-specific dependencies; for instance, a child transaction need not spend or have any relationship with the funds received in the parent transaction. We use the term *ancestor set* to refer to all transactions reachable via parent edges back in history, and *progeny* to refer to all children transactions and their offspring.

The central challenge in the maintenance of the DAG is to choose among *conflicting transactions*. The notion of conflict is application-defined and transitive, forming an equivalence relation. In our cryptocurrency application, transactions that spend the same funds (*double-spends*) conflict, and form a *conflict set* (shaded regions in Figure 5.4), out of which only a single one can be accepted. Note that the conflict set of a virtuous transaction is always a singleton.

Avalanche instantiates a Snowball instance for each conflict set. Whereas Snowball uses repeated queries and multiple counters to capture the amount of confidence built in conflicting transactions (colors), Avalanche takes advantage of the DAG structure and uses a transaction's progeny. Specifically, when a transaction T is queried, all transactions reachable from T by following the DAG

```
1: procedure INIT
2:     𝒯 ← ∅  // the set of known transactions
3:     𝒬 ← ∅  // the set of queried transactions
4: procedure ONGENERATETX(data)
5:     edges ← {T' ← T : T' ∈ PARENTSELECTION(𝒯)}
6:     T ← TX(data, edges)
7:     ONRECEIVETX(T)
8: procedure ONRECEIVETX(T)
9:     if T ∉ 𝒯 then
10:        if 𝒫_T = ∅ then
11:            𝒫_T ← {T}, 𝒫_T.pref ← T
12:            𝒫_T.last ← T, 𝒫_T.cnt ← 0
13:        else 𝒫_T ← 𝒫_T ∪ {T}
14:        𝒯 ← 𝒯 ∪ {T}, c_T ← 0.
```

Figure 5.1: Avalanche: transaction generation.

edges are implicitly part of the query. A node will only respond positively to the query if T and its entire ancestry are currently the preferred option in their respective conflict sets. If more than a threshold of responders vote positively, the transaction is said to collect a *chit*. Nodes then compute their *confidence* as the total number of chits in the progeny of that transaction. They query a transaction just once and rely on new vertices and possible chits, added to the progeny, to build up their confidence. Ties are broken by an initial preference for first-seen transactions. Note that chits are decoupled from the DAG structure, making the protocol immune to attacks where the attacker generates large, padded subgraphs.

5.2 Avalanche: Specification

Each correct node u keeps track of all transactions it has learned about in set $\mathcal{T}_u$, partitioned into mutually exclusive conflict sets $\mathcal{P}_T$, $T \in \mathcal{T}_u$. Since conflicts are

```
1: procedure AVALANCHELOOP
2:     while true do
3:         find T that satisfies T ∈ 𝒯 ∧ T ∉ Q
4:         𝒦 ← SAMPLE(𝒩\u, k)
5:         P ← Σ_{v∈𝒦} QUERY(v, T)
6:         if P ≥ α then
7:             c_T ← 1
8:             // update the preference for ancestors
9:             for T' ∈ 𝒯 : T' ←* T do
10:                if d(T') > d(𝒫_{T'}.pref) then
11:                    𝒫_{T'}.pref ← T'
12:                if T' ≠ 𝒫_{T'}.last then
13:                    𝒫_{T'}.last ← T', 𝒫_{T'}.cnt ← 1
14:                else
15:                    ++𝒫_{T'}.cnt
16:        else
17:            for T' ∈ 𝒯 : T' ←* T do
18:                𝒫_{T'}.cnt ← 0
19:        // otherwise, c_T remains 0 forever
20:        Q ← Q ∪ {T}    // mark T as queried
```

Figure 5.2: Avalanche: the main loop.

transitive, if T_i and T_j are conflicting, then they belong to the same conflict set, i.e. $\mathcal{P}_{T_i} = \mathcal{P}_{T_j}$. This relation may sound counter-intuitive: conflicting transitions have the *equivalence* relation, because they are equivocations spending the *same* funds.

We write $T' \leftarrow T$ if T has a parent edge to transaction T', The "$\overset{*}{\leftarrow}$"-relation is its reflexive transitive closure, indicating a path from T to T'. DAGs built by different nodes are guaranteed to be compatible, though at any one time, the two nodes may not have a complete view of all vertices in the system. Specifically, if $T' \leftarrow T$, then every node in the system that has T will also have T' and the same relation $T' \leftarrow T$; and conversely, if $T' \not\leftarrow T$, then no nodes will end up with $T' \leftarrow T$.

1: **function** ISPREFERRED(T)
2: **return** $T = \mathcal{P}_T.pref$
3: **function** ISSTRONGLYPREFERRED(T)
4: **return** $\forall T' \in \mathcal{T}, T' \overset{*}{\leftarrow} T$: ISPREFERRED(T')
5: **function** ISACCEPTED(T)
6: **return**

$$((\forall T' \in \mathcal{T}, T' \leftarrow T : \text{ISACCEPTED}(T')) \\ \wedge |\mathcal{P}_T| = 1 \wedge \mathcal{P}_T.cnt \geq \beta_1) \quad \texttt{// safe early commitment} \\ \vee (\mathcal{P}_T.cnt \geq \beta_2) \quad \texttt{// consecutive counter}$$

7: **procedure** ONQUERY(j, T)
8: ONRECEIVETX(T)
9: RESPOND(j, ISSTRONGLYPREFERRED(T))

Figure 5.3: Avalanche: voting and decision primitives.

Each node u can compute a confidence value, $d_u(T)$, from the progeny as follows: $d_u(T) = \sum_{T' \in \mathcal{T}_u, T \overset{*}{\leftarrow} T'} c_{uT'}$ where $c_{uT'}$ stands for the chit value of T' for node u. Each transaction initially has a chit value of 0 before the node gets the query results. If the node collects a threshold of α yes-votes after the query, the value $c_{uT'}$ is set to 1, otherwise remains 0 forever. Therefore, a chit value reflects the result from the one-time query of its associated transaction and becomes immutable afterwards, while $d(T)$ can increase as the DAG grows by collecting more chits in its progeny. Because $c_T \in \{0, 1\}$, confidence values are monotonic.

In addition, node u maintains its own local list of known nodes $\mathcal{N}_u \subseteq \mathcal{N}$ that comprise the system. For simplicity, we assume for now $\mathcal{N}_u = \mathcal{N}$, and elide subscript u in contexts without ambiguity.

Each node implements an event-driven state machine, centered around a query that serves both to solicit votes on each transaction and to notify other nodes of the existence of newly discovered transactions. In particular, when node u discovers a transaction T through a query, it starts a one-time query

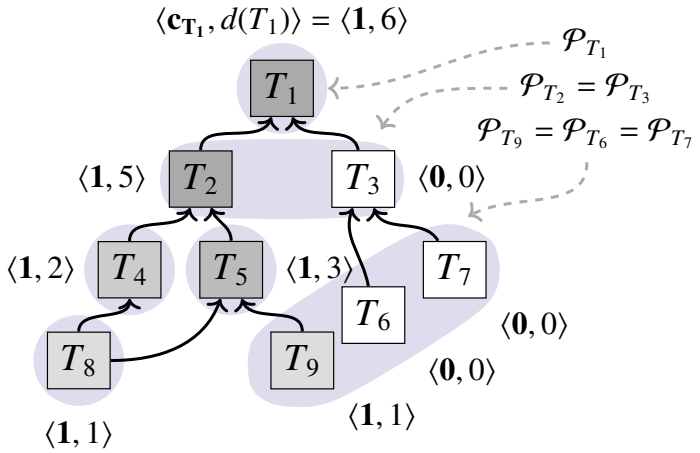


Figure 5.4: Example of ⟨chit, confidence⟩ values. Darker boxes indicate transactions with higher confidence values. At most one transaction in each shaded region will be accepted.

process by sampling k random peers and sending a message to them, after T is delivered via ONRECEIVETX.

Node u answers a query by checking whether each T' such that $T' \overset{*}{\leftarrow} T$ is currently preferred among competing transactions $\forall T'' \in \mathcal{P}_{T'}$. If every single ancestor T' fulfills this criterion, the transaction is said to be *strongly preferred*, and receives a yes-vote (1). A failure of this criterion at any T' yields a no-vote (0). When u accumulates k responses, it checks whether there are α yes-votes for T, and if so grants the chit (chit value $c_T \leftarrow 1$) for T. The above process will yield a labeling of the DAG with a chit value and associated confidence for each transaction T.

Figure 5.4 illustrates a sample DAG built by Avalanche. Similar to Snowball, sampling in Avalanche will create a positive feedback for the preference of a single transaction in its conflict set. For example, because T_2 has larger confidence than T_3, its descendants are more likely collect chits in the future compared to T_3.

Similar to Bitcoin, Avalanche leaves determining the acceptance point of a transaction to the application. An application supplies an ISACCEPTED predicate that can take into account the value at risk in the transaction and the chances of a decision being reverted to determine when to decide.

Committing a transaction can be performed through a *safe early commitment*. For virtuous transactions, T is accepted when it is the only transaction in its conflict set and has a confidence not less than threshold β_1. As in Snowball, T can also be accepted after a β_2 number of consecutive successful queries. If a virtuous transaction fails to get accepted due to a problem with parents, it could be accepted if reissued with different parents. Figure 5.1 shows how Avalanche entangles transactions. Because transactions that consume and generate the same UTXO do not conflict with each other, any transaction can be reissued with different parents.

Figure 5.2 illustrates the protocol main loop executed by each node. In each iteration, the node attempts to select a transaction T that has not yet been queried. If no such transaction exists, the loop will stall until a new transaction is added to $\mathcal{T}$. It then selects k peers and queries those peers. If more than α of those peers return a positive response, the chit value is set to 1. After that, it updates the preferred transaction of each conflict set of the transactions in its ancestry. Next, T is added to the set Q so it will never be queried again by the node. The code that selects additional peers if some of the k peers are unresponsive is omitted for simplicity.

Figure 5.3 shows what happens when a node receives a query for transaction T from peer j. First it adds T to $\mathcal{T}$, unless it already has it. Then it determines if T is currently strongly preferred. If so, the node returns a positive response to peer

j. Otherwise, it returns a negative response. Notice that in the pseudocode, we assume when a node knows T, it also recursively knows the entire ancestry of T. This can be achieved by postponing the delivery of T until its entire ancestry is recursively fetched. In practice, an additional gossip process that disseminates transactions is used in parallel, but is not shown in pseudocode for simplicity.

5.3 Multi-Input UTXO Transactions

In addition to the DAG structure in Avalanche, an *unspent transaction output* (UTXO) [119] graph that captures spending dependency is used to realize the ledger for the payment system. To avoid ambiguity, we denote the transactions that encode the data for money transfer *transactions*, while we call the transactions ($T \in \mathcal{T}$) in Avalanche's DAG *vertices*.

We inherit the transaction and address mechanisms from Bitcoin. At their simplest, transactions consist of multiple inputs and outputs, with corresponding redeem scripts. Addresses are identified by the hash of their public keys, and signatures are generated by corresponding private keys. The full scripting language is used to ensure that a redeem script is authenticated to spend a UTXO. UTXOs are fully consumed by a valid transaction, and may generate new UTXOs spendable by named recipients. Multi-input transactions consume multiple UTXOs, and in Avalanche, may appear in multiple conflict sets. To account for these correctly, we represent *transaction-input* pairs (e.g. $\texttt{In}_{a1}$) as Avalanche vertices. The conflict relation of transaction-input pairs is transitive because of each pair only spends one unspent output. Then, we use the conjunction of ISACCEPTED for all inputs of a transaction to ensure that no transaction

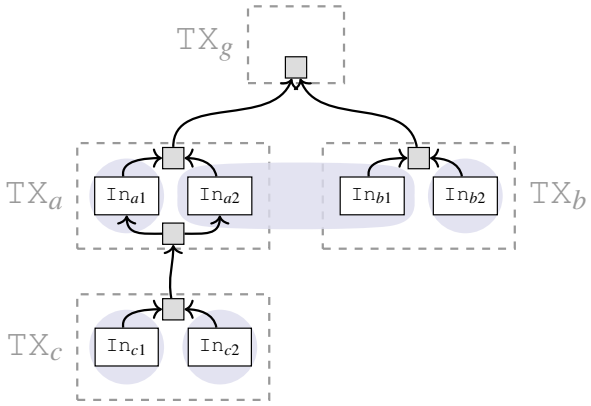


Figure 5.5: The underlying logical DAG structure used by Avalanche. The tiny squares with shades are dummy vertices which just help form the DAG topology for the purpose of clarity, and can be replaced by direct edges. The rounded gray regions are the conflict sets.

will be accepted unless all its inputs are accepted (Figure 5.5). In other words, a transaction is accepted only if all its transaction-input pairs are accepted in their respective Snowball conflict sets. Following this idea, we finally implement the DAG of transaction-input pairs such that multiple transactions can be batched together per query.

Optimizations We implement some optimizations to help the system scale. First, we use *lazy updates* to the DAG, because the recursive definition for confidence may otherwise require a costly DAG traversal. We maintain the current $d(T)$ value for each active vertex on the DAG, and update it only when a descendant vertex gets a chit. Since the search path can be pruned at accepted vertices, the cost for an update is constant if the rejected vertices have a limited number of descendants and the undecided region of the DAG stays at constant size. Second, the conflict set could be large in practice, because a rogue client can generate a large volume of conflicting transactions. Instead of keeping a container data structure for each conflict set, we create a mapping from each UTXO to the

preferred transaction that stands as the representative for the entire conflict set. This enables a node to quickly determine future conflicts, and the appropriate response to queries. Finally, we speed up the query process by terminating early as soon as the α threshold is met, without waiting for k responses.

DAG Compared to Snowball, Avalanche introduces a DAG structure that entangles the fate of unrelated conflict sets, each of which is a single-decree instance. This entanglement embodies a tension: attaching a virtuous transaction to undecided parents helps propel transactions towards a decision, while it puts transactions at risk of suffering liveness failures when parents turn out to be rogue. We can resolve this tension and provide a liveness guarantee with the aid of two mechanisms.

First we adopt an adaptive parent selection strategy, where transactions are attached at the live edge of the DAG, and are retried with new parents closer to the genesis vertex. This procedure is guaranteed to terminate with uncontested, decided parents, ensuring that a transaction cannot suffer liveness failure due to contested, rogue transactions. A secondary mechanism ensures that virtuous transactions with decided ancestry will receive sufficient chits. Correct nodes examine the DAG for virtuous transactions that lack sufficient progeny and emit no-op transactions to help increase their confidence. With these two mechanisms in place, it is easy to see that, at worst, Avalanche will degenerate into separate instances of Snowball, and thus provide the same liveness guarantee for virtuous transactions.

Unlike other cryptocurrencies [128] that use graph vertices directly as votes, Avalanche only uses a DAG for the purpose of batching queries in the under-

lying Snowball instances. Because confidence is built by collected chits, and not by just the presence of a vertex, simply flooding the network with vertices attached to the rejected side of a subgraph will not subvert the protocol.

5.4 Communication Complexity

Let the DAG induced by Avalanche have an expected branching factor of p determined by the parent selection algorithm. Given the β_1 and β_2 decision threshold, a transaction that has just reached the point of decision will have an associated progeny $\mathcal{Y}$. Let m be the expected depth of $\mathcal{Y}$. If we were to let the Avalanche network make progress and then freeze the DAG at a depth y, then it will have roughly py vertices/transactions, of which $p(y-m)$ are decided in expectation. Only pm recent transactions would lack the progeny required for a decision. For each node, each query requires k samples, and therefore the total message cost per transaction is in expectation $(pky)/(p(y-m)) = ky/(y-m)$. Since m is a constant determined by the undecided region of the DAG as the system constantly makes progress, message complexity per node is $O(k)$, while the total complexity is $O(kn)$.

5.5 Deployment Experience

Avalanche is deployed worldwide on over 800 nodes (fewer than 100 controlled directly by us) and in active use, confirming over 20K transactions per day. Anyone can join using our open source code. It has never experienced any safety failure. There was a temporary stall due to cross-chain block validation bug this

February, triggered by an unprecedented increase in use of our smart contract subsystem. The platform soon resumed after the fix. Due to the decentralized nature, we had a difficult time testing the fix: we did not own a majority of the stakes so new proposals by the altered nodes could be rejected as being "Byzantine", until community node owners caught up later.

5.6 Evaluation

We conduct our experiments on Amazon EC2 by running from hundreds (125) to thousands (2000) of virtual machine instances. We use `c5.large` instances, each of which simulates an individual node. AWS provides 2Gbps bandwidth, but Avalanche utilizes at most around 100 Mbps.

Our implementation supports two versions of transactions: one is the UTXO format, while the other uses the code directly from Bitcoin 0.16. Both supported formats use Bitcoin's secp256k1 crypto library and provide the same address format for wallets. All experiments use the customized format except for geo-replication, where results for both are given.

We simulate a constant flow of new transactions from users by creating separate client processes, each of which maintains separated wallets, generates transactions with new recipient addresses and sends the requests to Avalanche nodes. We use several such client processes to max out the capacity of our system. The number of recipients for each transaction is tuned to achieve average transaction sizes of around 250 bytes (1–2 inputs/outputs per transaction on average and a stable UTXO size), the current average transaction size of Bit-

coin. For efficiency, we batch up to 40 transactions during a query, but maintain confidence values at individual transaction granularity.

All reported metrics reflect end-to-end measurements taken from the perspective of all clients. That is, clients examine the total number of confirmed transactions per second for throughput, and, for each transaction, subtract the initiation timestamp from the confirmation timestamp for latency. Each throughput experiment is repeated for 5 times and standard deviation is indicated in each figure. As for security parameters, we pick $k = 10$, $\alpha = 0.8$, $\beta_1 = 11$, $\beta_2 = 150$, which yields an MTTF of $\sim 10^{24}$ years.

5.6.1 Throughput

We first measure the throughput of the system by saturating it with transactions and examining the rate at which transactions are confirmed in the steady state. For this experiment, we run Avalanche on 125 nodes with 10 client processes, each of which maintains 400 outstanding transactions at a given time.

As shown by the first group of bars in Figure 5.6, the system achieves 6851 transactions per second (tps) for a batch size of 20 and above 7002 tps for a batch size of 40. Our system is saturated by a small batch size comparing to other blockchains with known performance: Bitcoin batches several thousands of transactions per block, Algorand [79] uses 2–10 Mbyte blocks, i.e., 8.4–41.9K tx/batch and Conflux [101] uses 4 Mbyte blocks, i.e., 16.8K tx/batch. These systems are relatively slow in making a single decision, and thus require a very large batch (block) size for better performance. Achieving high throughput with small batch size implies low latency, as we will show later.

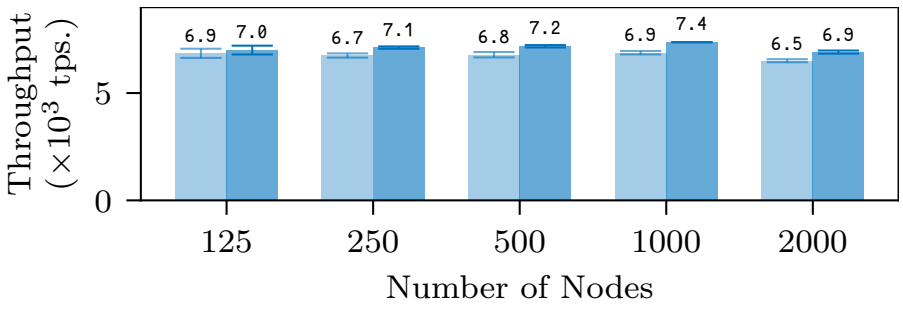


Figure 5.6: Throughput vs. network size. Each pair of bars is produced with batch size of 20 and 40, from left to right.

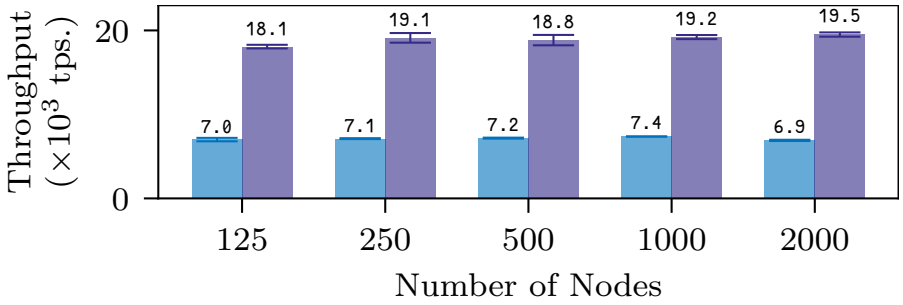


Figure 5.7: Throughput for batch size of 40, with (left) and without (right) signature verification.

5.6.2 Scalability

To examine how the system scales in terms of the number of nodes participating in Avalanche consensus, we run experiments with identical settings and vary the number of nodes from 125 up to 2000.

Figure 5.6 shows that overall throughput degrades about 1.34% to 6909 tps when the network grows by a factor of 16 to $n = 2000$. This degradation is minor compared to the growth of the network size. Note that the x-axis is logarithmic.

Avalanche acquires its scalability from three sources: first, maintaining a partial order that captures only the spending relations allows for more concurrency than a classical BFT replicated log that linearizes all transactions; second, the

lack of a leader naturally avoids bottlenecks; finally, the number of messages each node has to handle per decision is $O(k)$ and does not grow as the network scales up.

5.6.3 Cryptography Bottleneck

We next examine where bottlenecks lie in our current implementation. The purple bar on the right of each group in Figure 5.7 shows the throughput of Avalanche with signature verification disabled. Throughputs get approximately 2.6x higher, compared to the blue bar on the left. This reveals that cryptographic verification overhead is the current bottleneck of our implementation. This bottleneck can be addressed by offloading transaction verification to a GPU. Even without such optimization, 7K tps is far in excess of extant blockchains.

5.6.4 Latency

The latency of a transaction is the time spent from the moment of its submission until it is confirmed as accepted. Figure 5.8 tallies the latency distribution histogram using the same setup as for the throughput measurements with 2000 nodes. The x-axis is the time in seconds while the y-axis is the portion of transactions that are finalized within the corresponding time period. This figure also outlines the Cumulative Distribution Function (CDF) by accumulating the number of finalized transactions over time.

This experiment shows that most transactions are confirmed within approximately 0.3 seconds. The most common latencies are around 206 ms and variance

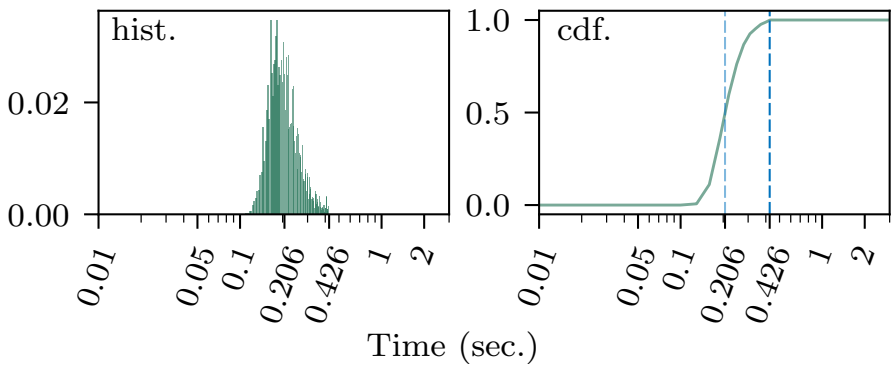


Figure 5.8: Transaction latency distribution for $n = 2000$. The x-axis is the transaction latency in log-scaled seconds, while the y-axis is the portion of transactions that fall into the confirmation time (normalized to 1). Histogram of all transaction latencies for a client is shown on the left with 100 bins, while its CDF is on the right.

is low, indicating that nodes converge on the final value as a group around the same time. The second vertical line shows the maximum latency we observe, which is around 0.4 seconds.

Figure 5.9 shows transaction latencies for different numbers of nodes. The horizontal edges of boxes represent minimum, first quartile, median, third quartile and maximum latency respectively, from bottom to top. Crucially, the experimental data show that median latency is more-or-less independent of network size.

5.6.5 Misbehaving Clients

We next examine how rogue transactions issued by misbehaving clients that double spend unspent outputs can affect latency for virtuous transactions created by honest clients. We adopt a strategy to simulate misbehaving clients where a fraction (from 0% to 25%) of the pending transactions conflict with

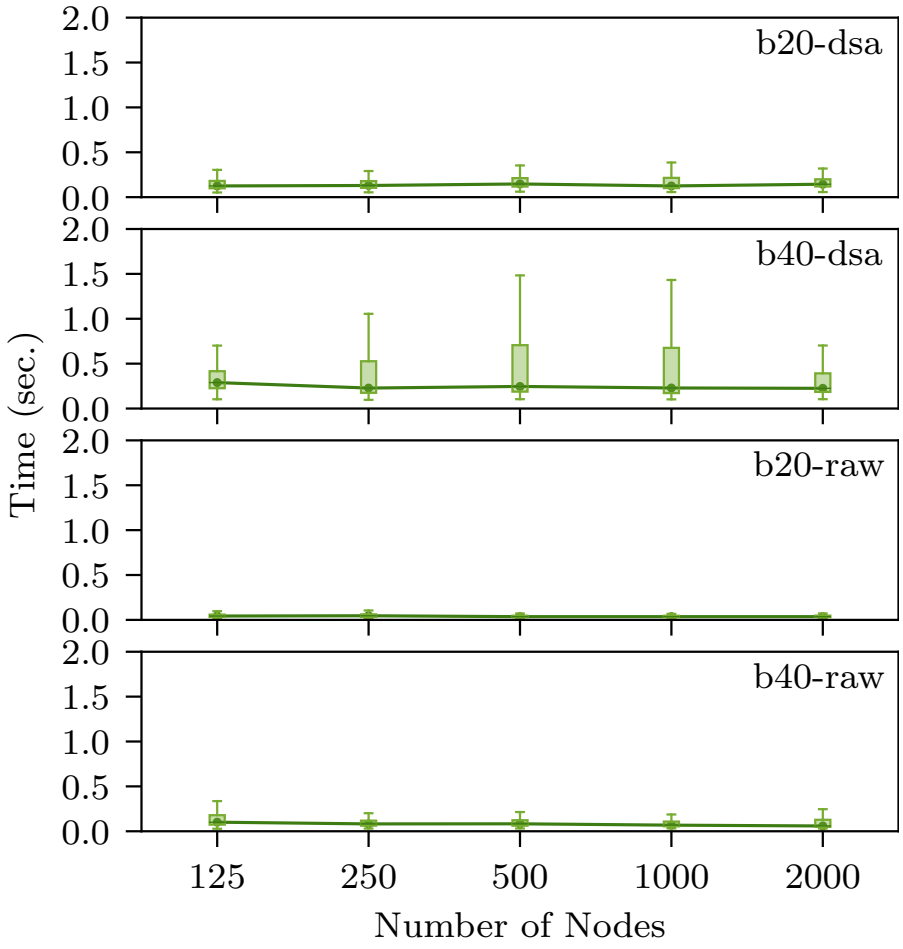


Figure 5.9: Transaction latency vs. network size. "b" indicates batch size and "raw" is the run without signature verification.

some existing ones. The client processes achieve this by designating some double spending transaction flows among all simulated pending transactions and sending the conflicting transactions to different nodes. We use the same setup with $n = 1000$ as in the previous experiments, and only measure throughput and latency of confirmed transactions.

Avalanche's latency is only slightly affected by misbehaving clients, as shown in Figure 5.10. Surprisingly, maximum latencies drop slightly when the percentage of rogue transactions increases. This behavior occurs because,

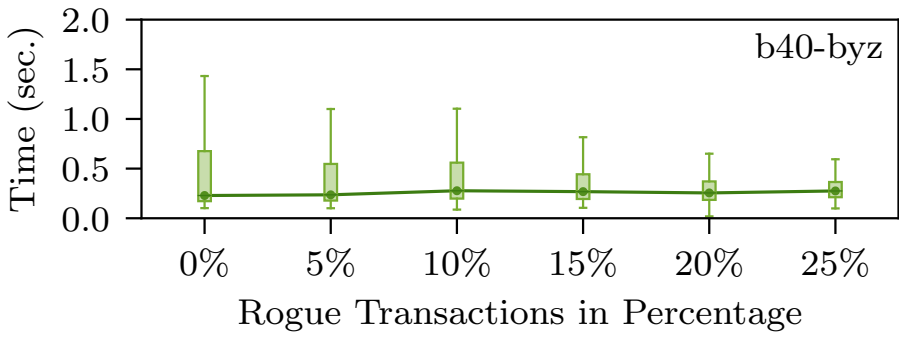


Figure 5.10: Latency vs. ratio of rogue transactions.

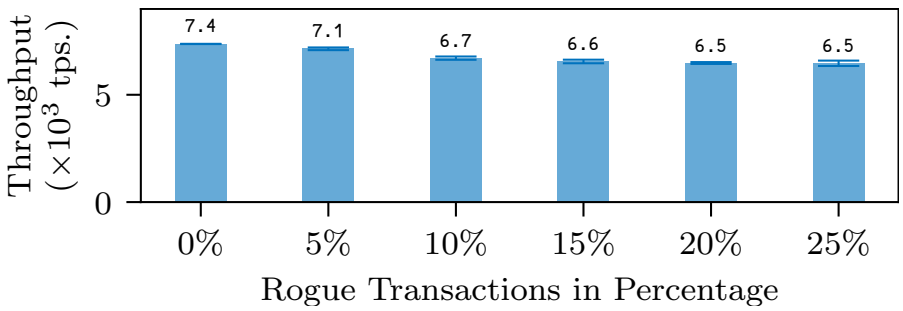


Figure 5.11: Throughput vs. ratio of rogue transactions.

with the introduction of rogue transactions, the overall *effective* throughput is reduced and thus alleviates system load. This is confirmed by Figure 5.11, which shows that throughput (of virtuous transactions) decreases with the ratio of rogue transactions. Further, the reduction in throughput appears proportional to the number of misbehaving clients, that is, there is no leverage provided to the attackers.

5.6.6 Geo-replication

The next experiment shows the system in an emulated geo-replicated scenario, patterned after the same scenario in prior work [79]. We selected 20 major cities near substantial numbers of reachable Bitcoin nodes, according to [28]. The

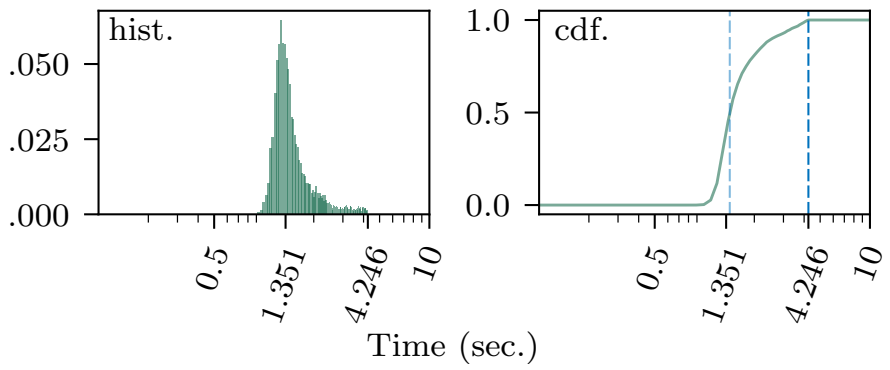


Figure 5.12: Latency histogram/CDF for n = 2000 in 20 cities.

cities cover North America, Europe, West Asia, East Asia, Oceania, and also cover the top 10 countries with the highest number of reachable nodes. We use the latency and jitter matrix crawled from [151] and emulate network packet latency in the Linux kernel using `tc` and `netem`. 2000 nodes are distributed evenly to each city, with no additional network latency emulated between nodes within the same city. Like Algorand's evaluation, we also cap our bandwidth per process to 20 Mbps to simulate internet-scale settings where there are many commodity network links. We assign a client process to each city, maintaining 400 outstanding transactions per city at any moment.

In this scenario, Avalanche achieves an average throughput of 3401 tps, with a standard deviation of 39 tps. As shown in Figure 5.12, the median transaction latency is 1.35 seconds, with a maximum latency of 4.25 seconds. We also support native Bitcoin code for transactions; in this case, the throughput is 3530 tps, with σ = 92 tps.

5.6.7 Comparison to Other Systems

Though there are seemingly abundant blockchain or cryptocurrency protocols, most of them only present a sketch of their protocols and do not offer practical implementation or evaluation results. Moreover, among those who do provide results, most are not evaluated in realistic, large-scale (hundreds to thousands of full nodes participating in consensus) settings.

Therefore, we choose Algorand and Conflux for our comparison. Algorand, Conflux, and Avalanche are all fundamentally different in their design. Algorand's committee-scale consensus algorithm is quorum-based Byzantine agreement, and Conflux extends Nakamoto consensus by a DAG structure to facilitate higher throughput, while Avalanche belongs to a new protocol family based on metastability. Additionally, we use Bitcoin [119] as a baseline.

Both Algorand and Avalanche evaluations use a decision network of size 2000 on EC2. We used shared `c5.large` instances, while Algorand used `m4.2xlarge`. These two platforms are similar except for a slight CPU clock speed edge for `c5.large`, which goes largely unused because our process only consumes 30% in these experiments. The security parameters chosen in our experiments guarantee a safety violation probability below 10^{-9} in the presence of 20% Byzantine nodes, while Algorand's evaluation guarantees a violation probability below 5×10^{-9} with 20% Byzantine nodes.

Neither Algorand nor Conflux evaluations take into account the overhead of cryptographic verification. Their evaluations use blocks that carry megabytes of dummy data and present the throughput in MB/hour or GB/hour unit. We use the average size of a Bitcoin transaction, 250 bytes, to derive their throughputs.

In contrast, our experiments carry real transactions and take all cryptographic overhead into account.

The throughput is 3-7 tps for Bitcoin, 874 tps for Algorand (with 10 Mbyte blocks), 3355 tps for Conflux (in the paper it claims 3.84x Algorand's throughput under the same settings).

In contrast, Avalanche achieves over 3400 tps consistently on up to 2000 nodes without committee or PoW. As for latency, a transaction is confirmed after 10–60 minutes in Bitcoin, around 50 seconds in Algorand, 7.6–13.8 minutes in Conflux, and 1.35 seconds in Avalanche.

Avalanche performs much better than Algorand in both throughput and latency because Algorand uses a verifiable random function to elect committees, and maintains a totally-ordered log while Avalanche establishes only a partial order. Although Algorand's leadership is anonymous and changes continuously, it is still leader-based which could be the bottleneck for scalability, while Avalanche is leader-less.

Avalanche has similar throughput to Conflux, but its latency is 337–613x better. Conflux also uses a DAG structure to amortize the cost for consensus and increase throughput, however, it is still rooted in Nakamoto consensus (PoW), making it unable to have instant confirmation.

In a blockchain system, one can usually improve throughput at the cost of latency through batching. The real bottleneck of the performance is the number of decisions the system can make per second, and this is fundamentally limited by either Byzantine Agreement (BA^*) in Algorand and Nakamoto consensus in Conflux.

5.7 Related Work

Bitcoin [119] is a cryptocurrency that uses a blockchain based on PoW to maintain a ledger of UTXO transactions. While techniques based on PoW [17, 70], and even cryptocurrencies with minting based on PoW [134, 149], have been explored before, Bitcoin was the first to incorporate PoW into its consensus process. Unlike more traditional BFT protocols, Bitcoin has a probabilistic safety guarantee and assumes honest majority computational power rather than a known membership, which in turn has enabled an internet-scale permissionless protocol. While permissionless and resilient to adversaries, Bitcoin suffers from low throughput (~3 tps) and high latency (~5.6 hours for a network with 20% Byzantine presence and 2^{-32} security guarantee). Furthermore, PoW requires a substantial amount of computational power that is consumed only for the purpose of maintaining safety.

Countless cryptocurrencies use PoW to maintain a distributed ledger. Like Bitcoin, they suffer from inherent scalability bottlenecks. Several proposals for protocols exist that try to better utilize the effort made by PoW. Bitcoin-NG [73] and the permissionless version of Thunderella [125] use Nakamoto-like consensus to elect a leader that dictates writing of the replicated log for a relatively long time to provide higher throughput. Moreover, Thunderella provides an optimistic bound that, with 3/4 honest computational power and an honest elected leader, allows transactions to be confirmed rapidly. ByzCoin [91] periodically selects a small set of participants and then runs a PBFT-like protocol.

Protocols based on Byzantine agreement [97, 126] typically make use of quorums and require precise knowledge of membership. PBFT [43, 44], a well-

known representative, requires a quadratic number of message exchanges in order to reach agreement. The Q/U protocol [5] and HQ replication [54] use a quorum-based approach to optimize for contention-free cases of operation to achieve consensus in only a single round of communication. However, although these protocols improve on performance, they degrade very poorly under contention. Zyzzyva [92] couples BFT with speculative execution to improve the failure-free operation case. Past work in permissioned BFT systems typically requires at least $3f + 1$ replicas. CheapBFT [86] leverages trusted hardware components to construct a protocol that uses $f + 1$ replicas.

Other work attempts to introduce new protocols under redefinitions and relaxations of the BFT model. Large-scale BFT [137] modifies PBFT to allow for arbitrary choice of number of replicas and failure threshold, providing a probabilistic guarantee of liveness for some failure ratio but protecting safety with high probability. In another form of relaxation. Zeno [142] introduces a BFT state machine replication protocol that trades consistency for high availability. More specifically, Zeno guarantees eventual consistency rather than linearizability, meaning that participants can be inconsistent but eventually agree once the network stabilizes. By providing an even weaker consistency guarantee, namely fork-join-causal consistency, Depot [107] describes a protocol that guarantees safety under $2f + 1$ replicas.

NOW [81] uses sub-quorums to drive smaller instances of consensus. The insight of this paper is that small, logarithmic-sized quorums can be extracted from a potentially large set of nodes in the network, allowing smaller instances of consensus protocols to be run in parallel.

Snow White [56] and Ouroboros [89] are some of the earliest provably secure PoS protocols. Ouroboros uses a secure multiparty coin-flipping protocol to produce randomness for leader election. The follow-up protocol, Ouroboros Praos [57] provides safety in the presence of fully adaptive adversaries. Honey-Badger [115] provides good liveness in a network with heterogeneous latencies.

Tendermint [35, 36] rotates the leader for each block and has been demonstrated with as many as 64 nodes. Ripple [139] has low latency by utilizing collectively-trusted sub-networks in a large network. The Ripple company provides a slow-changing default list of trusted nodes, which renders the system essentially centralized. HotStuff [154, 155] improves the communication cost from quadratic to linear and significantly simplifies the protocol specification, although the leader bottleneck still persists. Facebook uses HotStuff as the core consensus for its Libra project. In the synchronous setting, inspired by HotStuff, Sync HotStuff [10] achieves consensus in 2Δ time with quadratic cost and unlike other lock-steped synchronous protocols, it operates as fast as network propagates. Stellar [110] uses Federated Byzantine Agreement in which *quorum slices* enable heterogeneous trust for different nodes. Safety is guaranteed when transactions can be transitively connected by trusted quorum slices. Algorand [79] uses a verifiable random function to select a committee of nodes that participate in a novel Byzantine consensus protocol.

Some protocols use a Directed Acyclic Graph (DAG) structure instead of a linear chain to achieve consensus [21, 25, 143–145]. Instead of choosing the longest chain as in Bitcoin, GHOST [144] uses a more efficient chain selection rule that allows transactions not on the main chain to be taken into consideration, increasing efficiency. SPECTRE [143] uses transactions on the DAG to vote

recursively with PoW to achieve consensus, followed up by PHANTOM [145] that achieves a linear order among all blocks. Like PHANTOM, Conflux also finalizes a linear order of transactions by PoW in a DAG structure, with better resistance to liveness attack [101]. Avalanche is different in that the voting result is a one-time chit that is determined by a query without PoW, while the votes in PHANTOM or Conflux are purely determined by PoW in transaction structure. Similar to Thunderella, Meshcash [25] combines a slow PoW-based protocol with a fast consensus protocol that allows a high block rate regardless of network latency, offering fast confirmation time. Hashgraph [21] is a leader-less protocol that builds a DAG via randomized gossip. It requires full membership knowledge at all times, and, similar to the Ben-Or [24], it requires exponential messages [16, 41] in expectation.

CHAPTER 6

CEDRUSDB: PERSISTENT KEY-VALUE STORE WITH MEMORY-MAPPED LAZY-TRIE

The two BFT replication protocols from the previous chapters cover a wide range of blockchain infrastructures and can offer reasonable performance in practice. As replication of state machine transitions becomes cheap and simple, the next bottleneck in a blockchain that supports verifiable, generic computation may shift to the storage of the persistent, historical states. We noticed the scalability issue caused by log compaction in LevelDB while implementing the Avalanche prototype. Inspired by hash-tries used in modern programming language runtimes and the application-level Merkle Patricia trees in Ethereum, we consider if it is possible to use a hashed, radix tree-like data structure to persist unsorted items in the low-level storage. In this chapter, we design and evaluate a persistent, disk-indexed key-value store named CedrusDB [156] that could be used by blockchains and other general applications.

6.1 The Lazy-Trie Data Structure

The design of CedrusDB is inspired by memory-mapped key-value stores like LMDB [53]. Instead of using B^+-tree variants or other tree structures that require complex operations to move nodes across sibling subtrees for a logarithmic tree height, we propose *lazy-trie*, a trie-tree structure tailored for persistent storage. In this section, we describe the lazy-trie by progressively adding its crucial elements, along with a discussion of its properties and their implications for storage.

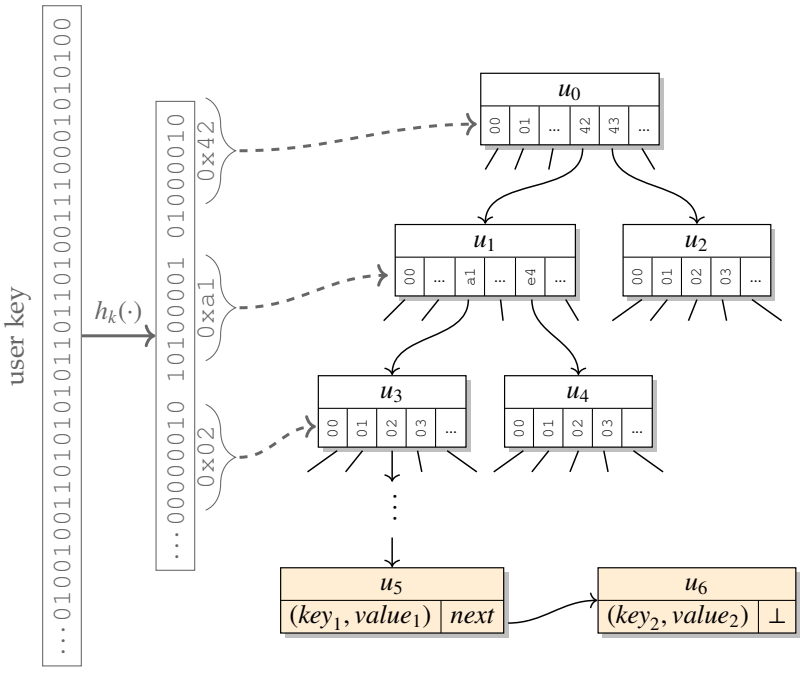


Figure 6.1: Structure of a hash-trie with chained leaf nodes.

6.1.1 Hash-Trie for Persistent Storage

A *trie* is a tree structure that encodes all prefix paths of inserted key strings. Each key is treated as a sequence of consecutive fixed-width *characters*. A trie stores paths of edges representing character sequences, collapsing all shared prefix into a tree topology.

A *hash-trie* is a trie indexed by a hash. CedrusDB uses a strong (well-distributed) and fast hash function (Section 6.2) to map an arbitrary key string given by the user to a 256-bit hash. It then partitions the hash into equal-length characters. In the trie, a *tree node* consists of a character-indexed array of pointers to its child nodes and a pointer referring back to its parent node. We define the *height* of a given node to be the number of its ancestors. Each leaf node maintains a linked list of user key-value pairs. Figure 6.1 depicts the basic structure

of a hash-trie, where the user key key_1 and key_2 in data node u_5 and u_6 have the same hash prefixed by `0x42a102`. The size of the hash should be chosen so that the probability of such collisions is small.

The insertion algorithm starts from the root node. It first visits the child indexed by the first character of the key hash, proceeds to the next child by the second character, and recursively walks down the trie until the entire key sequence is consumed. Child nodes are created as needed. At the leaf node, the algorithm adds a *data node* containing the original key-value data of the user to the linked list.

Lookup is similar to insertion. When reaching the leaf node, the linked list is scanned doing a full comparison using the original, unhashed user key to locate the value.

There are some similarities between the hash-trie and the B^+-tree data structures. First, both are balanced, n-ary tree structures. In practice, a typical B^+-tree has a branching factor of several hundreds. As we show later in the evaluation, a good choice for the branching factor for the hash-trie is hundreds of children per node as well. Second, both B^+-trees and hash-tries only store user data at the leaves—intermediate nodes only contain metadata for indexing.

That being said, the structures have important differences. The height of a B^+-tree is logarithmic in the number of keys. When inserting data, a B^+-tree has to constantly adjust its topology across sibling subtrees to maintain tree balance. In a hash-trie, the path to the leaf node for a specific key is *static*: there is no reorganization of this path when other data is inserted or deleted. This significantly reduces the I/O cost of maintaining the internal index structure

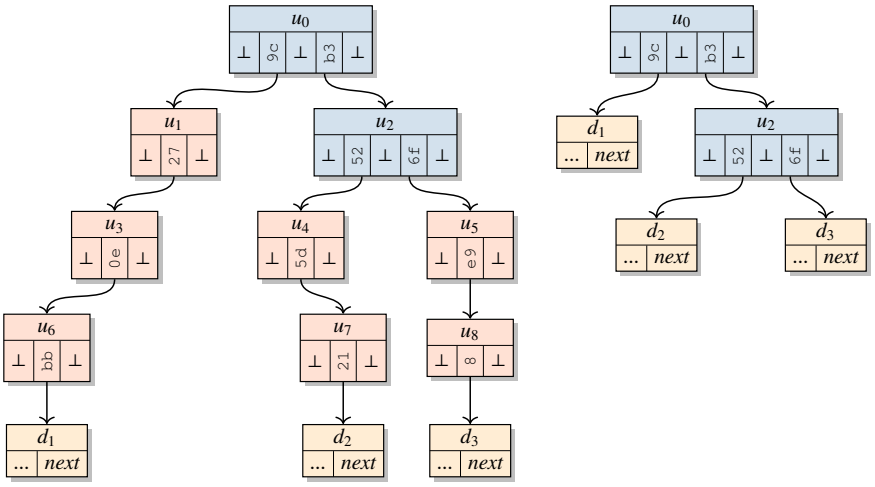


Figure 6.2: The tree structure without (left) and with (right) the path compression.

and simplifies concurrent access or modification to the tree, by sacrificing the support for range queries. Finally, all tree nodes of a hash-trie have an identical size and the same storage footprint, which simplifies the storage maintenance and reduces fragmentation. B$^+$-tree nodes have a variable number of children, bounded by the branching factor.

A disadvantage of the hash-trie design is that operations always need to visit a fixed number of tree nodes even if the data store is small. Also, the utilization of child tables in tree nodes can be low, potentially resulting in significant write amplication. We next show, in two steps, how to improve upon the hash-trie structure to build a practical, dynamically grown *lazy-trie* with near-optimal height and small storage overhead by utilizing statistical properties of the hash function.

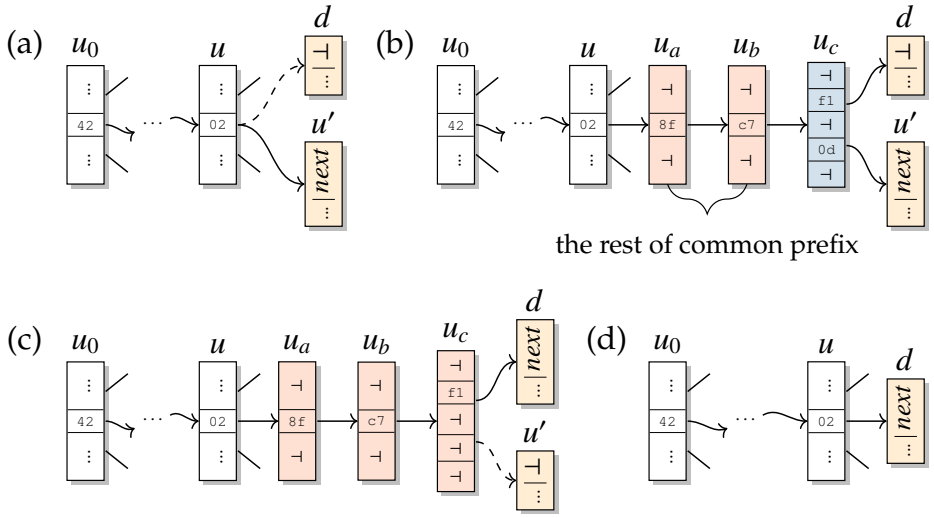


Figure 6.3: The bookkeeping operations required by lazy-trie. (a)(b) show a split operation, when a new data node d is inserted. (b)(c)(d) show a merge operation when the parent node u_c only has one child after the other child is removed.

6.1.2 Path Compression

Let the branching factor be b. Then two keys have b^{-h} probability of sharing the same prefix of length h. The exponentially decreasing probability means that, as one descends into the tree, it becomes less likely to have forks in the tree. This led us to compress the suffix of paths in the hash-trie and only unroll the compressed path when necessary. Data nodes remain at the leaves, so the collision rate remains unchanged. Figure 6.2 illustrates the new design. Compared to radix trees, the difference is that we only compress the *suffixes* of paths—internal paths may still have nodes with only one child.

Whenever inserting a new node, we lazily create the minimum number of intermediate missing nodes just to distinguish data nodes with different hashes. During the insertion, we first follow the tree structure as usual. If we reach a data node that is supposed to be a tree node, then we insert a tree node, uncompressing the path (see Figure 6.3).

The lookup algorithm stays essentially the same: CedrusDB traverses the lazy-trie according to the key hash until it reaches a data node, then it scans through the data nodes. In the delete algorithm, when a data node is removed, a *merge* operation checks whether it is the only child of its parent node. If so, it collapses the non-forking path repeatedly until reaching an ancestor having more than one child.

Unlike insertion or deletion in other tree structures, which require complex recursive reorganization, there is at most one non-recursive split for each insertion and one non-recursive merge for each deletion. In a split, a simple path of intermediate nodes is created as a chain, while, in a merge, the longest non-forking path is collapsed into a direct parent reference. These operations only happen on a *single path* of insertion and deletion, so they do not interfere with any other siblings or their subtrees.

As a result, lazy-trie dynamically adjusts the tree height depending on how frequently the prefixes of inserted key hashes conflict. The hash function offers uniformly distributed key hashes regardless of the order of insertion, statistically balancing the tree with a height of approximately $\log_b n$, where n is the number of keys. Since the probability of a shared prefix decreases exponentially with the length of the prefix, in expectation there will be few missing intermediate nodes in a split operation.

6.1.3 Sluggish Splitting

The lazy-trie structure has an average path length that grows logarithmically with the number of keys (Figure 6.4, top, $s = 1$). However, due to the probabilis-

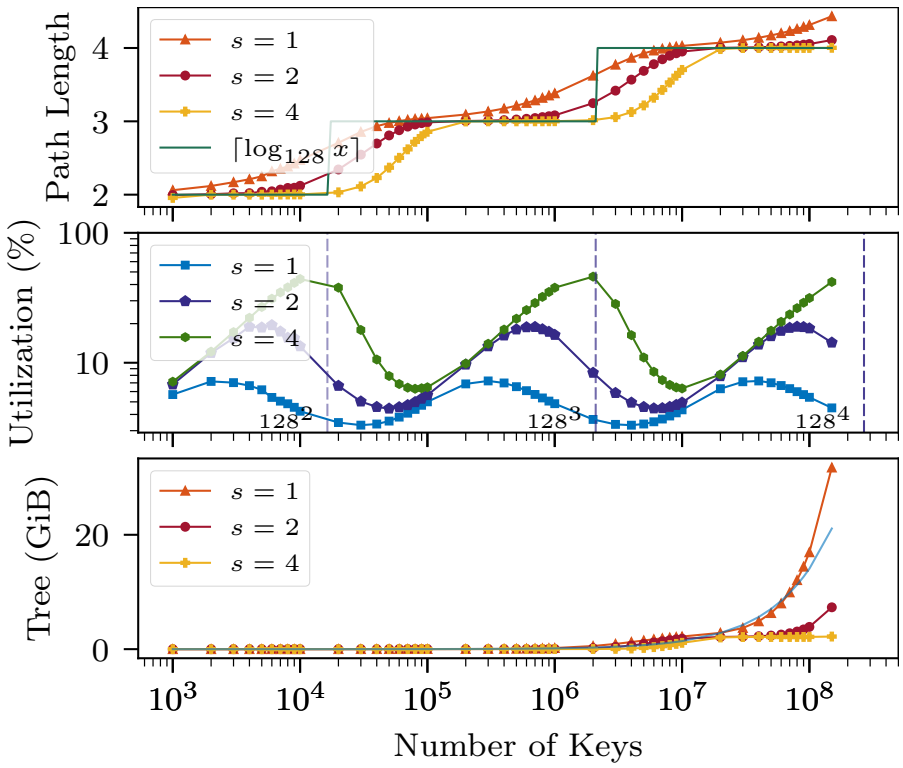


Figure 6.4: Average path length, child table utilization, and meta-data storage footprint for a 128-ary lazy-trie as a function of the number of keys, with different sluggishness values. The solid blue line in the bottom graph shows the user data footprint for 23-byte keys and 128-byte values.

tic nature, the paths for different keys can have significantly different lengths and the height of the trie (the maximum path length) can be significantly larger than the average path length. Moreover, the child tables in the leaf nodes tend to be mostly empty, leading to significant space underutilization because the majority of the space in a node is its child table (Figure 6.4, middle, $s = 1$). The problem is that the lazy-trie must guarantee that keys with different hashes cannot be in the same linked list.

To reduce variance in path lengths and improve utilization, we allow keys with different hashes to share the same linked list. We define *sluggishness* to be the maximum number of hash values allowed in a linked list of data nodes. The

sluggishness bounds the worst-case scanning time in the linked list. A *sluggish lazy-trie* will only split the path when a linked list overflows. If so, the linked list will be replaced with a leaf node, and all data nodes in the linked list will be redistributed into the child table of the new leaf node. The redistribution is recursive and continues until the sluggishness constraint is met. Figure 6.5 shows an example of the steps in redistribution.

To demonstrate how sluggish splitting mitigates the problems of high path length variation and low utilization, Figure 6.4 shows the average path length and child table utilization as the data store grows in size for different levels of sluggisness s. The periodic change of utilization is due to the statistical growth of the tree height. The bottom graph shows the number of bytes used by tree nodes as a function of the number of keys. With a sluggishness $s = 4$, the storage overhead is significantly reduced compared to no sluggish splitting ($s = 1$).

We only consider sluggishness in splitting and otherwise retain the original merge algorithm. The only downside to not chaining leaves back into linked lists after deletion is that the tree metadata do not optimally shrink back after items are removed. However, left-over nodes will still be re-used when new items are inserted. We confirmed that the tree footprint does not increase over time by running random insertion/deletion (50% for each) experiments with 1 billion operations on an initial store of 100 million items.

6.2 System Design and Implementation

In this section, we show how a lazy-trie can be used to build CedrusDB, a high-performance key-value store. CedrusDB has been fully implemented in ~8K

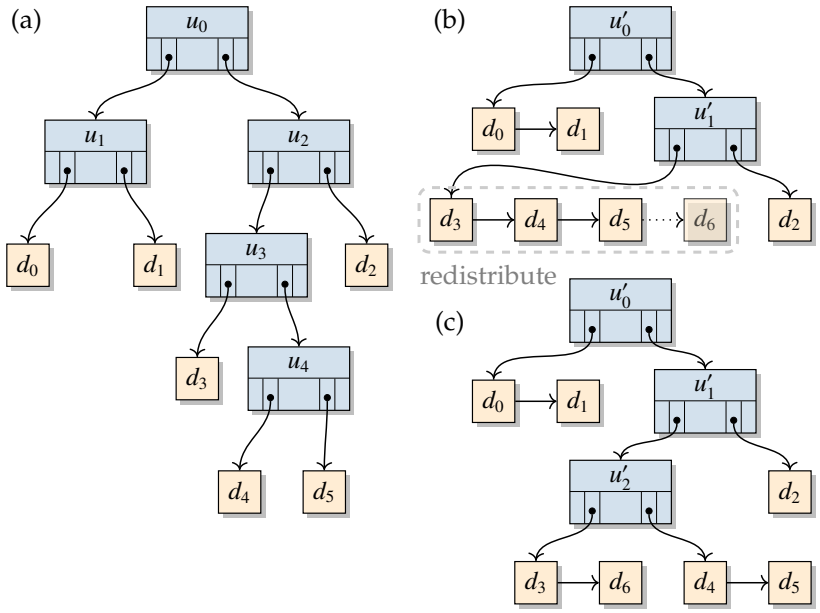


Figure 6.5: (a) a trie without sluggishness ($s = 1$). (b) in a sluggish lazy-trie, an insertion (d_6) can lead to overflowing the linked list of data nodes. (c) for a maximum sluggishness of 3, the data nodes are redistributed.

lines of pure Rust. Rust is a modern systems programming language that provides statically checked memory- and thread-safety guarantees [15, 61, 147]. In addition to basic constructs offered by the standard library, it allows programmers to customize their building blocks with different safety guarantees [62]. For CedrusDB, we explored how to separate the lazy-trie algorithm from the underlying storage management. We also made use of native functionalities of Linux, some of which may not be available on other operating systems, such as `io_submit` for kernel-based asynchronous IO (AIO).

6.2.1 Logical Spaces

In some key-value stores, like LMDB [53], the entire store is always mapped in memory and the maximum size has to be predetermined at its creation. Ce-

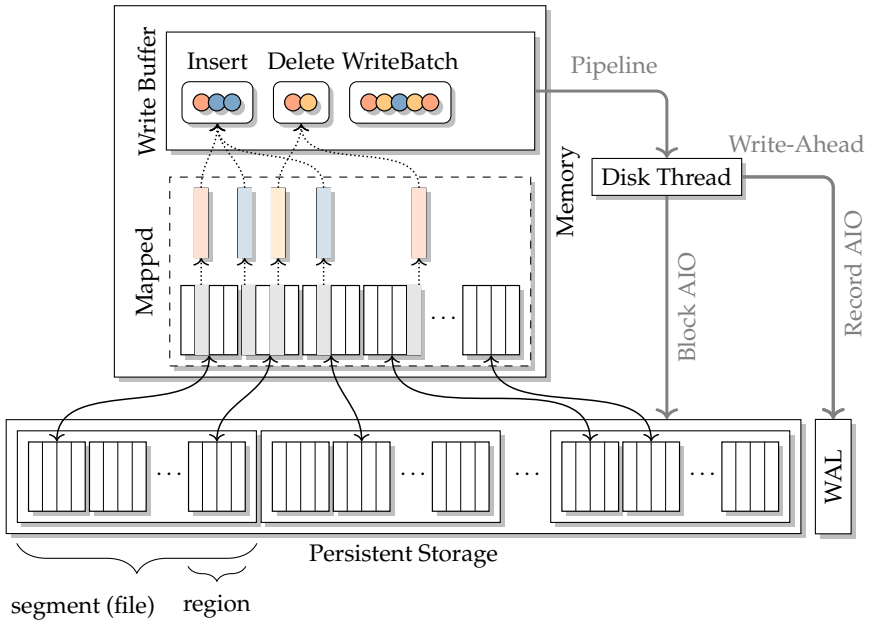


Figure 6.6: CedrusDB storage hierarchy with mapped memory and write buffer. The smallest rectangle represents a page.

drusDB supports a dynamically growing data set that may not all fit in memory. To support this, it has *logical spaces*, one for each type of objects. A logical space is a 64-bit four layer virtual address space. A *logical address* is a 64-bit unsigned integer subdivided into four parts: a segment number, a region number, a page number, and a page offset. CedrusDB associates a file with each segment. Each region is either fully mapped or unmapped. When mapped, it can be accessed like ordinary memory, that is, regions are memory-mapped that allow zero-copy read access. Figure 6.6 shows the organization of storage units with different granularities.

Objects cannot span across regions. Regions can only be accessed by mapping them, and thus the number of mapped regions is effectively the cache size, blurring the boundary between the "cache-based" approach, used by stores like LevelDB and RocksDB, and the "memory-based" approach by stores like LMDB

and Memcached. Ideally, the performance is optimal when regions can remain mapped as long as they are in active use, which is the main focus of this chapter.

To map regions, we use `mmap`. While in theory we could let the kernel keep track of dirty pages and write them back to the underlying segment files, we found that this solution, while simple, did not provide good performance even when `madvise` is used. The kernel ends up writing the same, actively modified pages repeatedly, thus incurring prohibitively high write cost. Moreover, when the bounded kernel buffer of pending writes is full, the kernel slows down store instructions (such as x86 `mov`) made to the virtual memory space, resulting in performance that is hard to predict.

The storage architecture of CedrusDB is therefore hybrid. We map the regions in memory but keep track of our own write buffer for page writes. Doing so also benefits write-ahead logging (Section 6.2.4).

Logical spaces allow the lazy-trie algorithm to operate transparently as if the entire data structure is in memory. CedrusDB maintains four logical spaces:

1. *trie space*: tree nodes in use;
2. *trie free list*: a stack of pointers to unused tree nodes;
3. *data space*: data nodes in use;
4. *data free list*: an array of descriptors tracking the unused portion of data space.

Tree nodes have a fixed size, and thus the trie space contains an array of tree nodes. The trie free list contains a stack of indexes into the tree node array. To

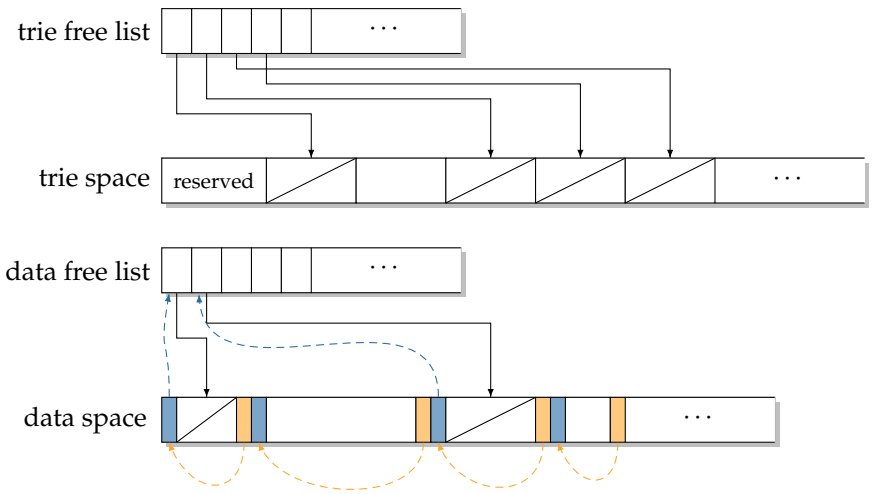


Figure 6.7: Four logical spaces used in CedrusDB.

free a tree node, its index is pushed onto the stack. To allocate a tree node, its index is popped from the stack.

In the data space, in-use and free data nodes of different sizes are stored continuously with a header at the front and a footer at the end. The footer of a data node is right before the header of the next data node and contains the size of the data node. It supports merging of two adjacent free data nodes (see below). Instead of maintaining a doubly-linked list of free data nodes like the free list in glibc's `malloc`, CedrusDB maintains a separate, unsorted array of *hole descriptors* for better locality and therefore I/O efficiency (see Figure 6.7). Each descriptor points to a free data node, while the header of a free data node points back to its descriptor.

CedrusDB adopts a *next-fit allocation policy* [84, 90], by indexing into the array of hole descriptors. An index points to the last visited descriptor and gets wrapped around when it hits the last item in the array. The index pointer, together with the trie root pointer and other tail pointers that indicate the bound-

ary of logical spaces, are all stored within the first (reserved) 4KBytes page of the trie space, before the entire tree node array.

6.2.2 Memory Abstraction and Layout

To benefit from Rust's memory-safety and thread-safety guarantees, we crafted our own memory-mapped object abstraction that fits in Rust's idiom and thus utilizes its type checker. Rust does not come natively with a type-safe wrapper or primitive for memory-mapping. Although nix [120], a popular Rust library, provides related functions, they are exposed as Rust function signatures of the original POSIX interface in C. They are unsafe because the compiler makes normal assumptions about memory management and variable life-times without making accommodations for memory-mapped address space.

Space and Objects

The `MMappedSpace` struct is the core of implementing the logical space abstraction (Section 6.2.1). It can be created from file handles and exposes safe methods.

We use an opaque struct to represent a pointer into logical space so that either it can be reconstructed from persistent storage (like a pointer to an existing tree node) or allocated through `MMappedSpace`. The reason we need an abstraction for a pointer is two-fold. First, addresses in logical spaces do not directly correspond to virtual memory addresses even if the corresponding region is mapped. Second, using an integer indexing into logical space would be

unsafe. Given a typed object pointer, `ObjPtr<T>`, it can dereference the pointer to the correspondingly typed object handle, `ObjRef<T>`. This allows manipulating the actual object typed `T` available in memory as if there is no memory-mapping. (It will auto-dereference to `&mut T`, which is an ordinary mutable access to a variable typed `T`.) The object handle is accessible throughout its lifetime by pinning the affected region in memory.

Tree Metadata

A lazy-trie tree node consists predominantly of a fixed-size array of pointers to its children.Two bitmasks indicate the validity and type of the pointer in a particular slot. To obtain child information, `chd_mask` is first examined to determine whether the slot contains a valid pointer. If so, `data_mask` specifies whether the pointer is to another tree node (`struct Node`) or a data node (`struct Data`). Both bitmasks are accessed using bitwise arithmetic. On x86-64 we use inline assembly with BMI quadword instructions such as `bsfq` and `bzhiq` for optimal performance.

We use the `#[repr(C)]` compiler directive to ensure a C-struct layout, guaranteeing that all memory-mapped objects can be correctly accessed even if the Rust compiler changes the default layout of `struct`. Unfortunately we cannot take advantage of the feature-rich Rust `enum` type. Thus, `ChdRaw` (union type for each child pointer) is hidden to other parts of the CedrusDB, which access the trie data structure through safe enum structs and methods.

User Data

We use Google's HighwayHash [13] to hash keys. For each user key-value pair, we use an object of `Data` struct to hold the precomputed HighwayHash of the original arbitrary-length key in `hkey`, and size information for the original key and value in `key_size` and `val_size`. The actual user key and value are placed directly after the Rust structure. To support keys with colliding hashes and sluggish splitting, `Data` objects are chained together into a singly-linked list using the `next` pointer. We use `madvise` [1] to request that the kernel pre-fetches the memory pages storing data objects.

6.2.3 Disk I/O

As described in Section 6.2.1 and shown in Figure 6.6, CedrusDB uses a separate write buffer to serialize all changes and schedule them as *block writes* to the physical storage device. Each modification made to the internal lazy-trie data structure first writes to memory.[1] The memory thus always reflects the latest changes. To update the underlying storage, the same write is also sent to a *disk thread* via a bounded buffer.

The changes generated to the in-memory data structures are short and frequent, which is not optimal for secondary storage. The disk thread therefore aggregates updates into blocks. The Linux ext4 filesystem has the same block size as the page size, so CedrusDB uses 4KBytes blocks.

[1] Due to limitations of `mmap`, we map the memory in copy-on-write mode even though we do not require that the original contents is saved.

While the lazy-trie data structure does not require log compaction such as used in LSMs and is arguably simpler to maintain than B^+-trees, it suffers potentially from non-optimal locality and random writes when pointers need to be updated. To optimize storage updates, the CedrusDB disk thread uses Linux native Asynchronous I/O (AIO). Not to be confused with the POSIX AIO standard offered by glibc [129], Linux AIO performs concurrent writes if possible. We access AIO through `libaio`, a thin C ABI wrapper that is available on mainstream Linux distributions. AIO allows us to asynchronously manage reads and writes in a non-blocking style.

Because CedrusDB supports concurrent operations on the lazy-trie, it is possible that multiple user threads make changes to the same page (block). In CedrusDB it is the disk thread's responsibility to obtain the consistent state of a block from a file if it is not already available in its cache, rather than copying the content from the mapped memory worked on by user threads, as there may be a data race. The disk thread can schedule such an infrequent block read while at the same time it can schedule other writes without being blocked.

6.2.4 Crash Recovery

CedrusDB utilizes write-ahead logging (WAL) to achieve atomicity and durability [116]. In the disk thread I/O pipeline, disk write records are first fed into a WAL worker. To better manage asynchronous I/O events, we implemented a library on top of `libaio` that manages I/O with *futures*. We then encapsulated the WAL logic into another library that schedules record writes via its own I/O manager handle. The disk thread pushes a vector of records to the WAL worker

using the library, which immediately returns a future object that gets resolved when the record write is complete. The disk thread schedules block writes using its own I/O manager. Likewise, when the future gets resolved upon completion, the thread notifies the WAL worker that specific records can be pruned.

Similar to RocksDB, CedrusDB WAL records are grouped into fixed sized chunks. Each chunk contains a flag indicated if it is continued by the following one. CedrusDB operations may generate small writes to various locations in the logical space. Encoding high-level descriptions of operations directly as records does not work well for redos. Instead, CedrusDB encodes all actual low-level disk writes in each record. To avoid write amplification, it uses a compact format for records. A record is the concatenation of several subrecords, each of which encodes a single update made to a logical space.

6.2.5 Concurrent Access

The lazy-trie design does not require tree re-balancing operations, simplifying concurrent access. Because walks down the tree diverge exponentially fast, gains from concurrent access can be significant.

Tree-Walk Parallelism

Locating the leaf node does not change the trie structure, while insertion or deletion only makes changes starting from a leaf node. We assign a reader-writer lock to each tree node to control any access that goes through that node.

First consider lookup operations. For each node visited on the path to the leaf node, a thread acquires the read lock. Knowing that once the thread holds a read lock, no concurrent updates can happen to the node or any node below, it is safe for the thread to release the lock on the parent node, allowing concurrent updates to other parts of the trie.

In theory, insertions could also obtain read locks similar to lookups, until the thread needs to update a node. At that time, the thread would have to upgrade its read lock to a write lock. In practice, Rust does support an upgradable reader-writer lock, but it only allows at most one thread to hold an upgradable read lock at a time (while other threads may concurrently hold a regular read lock). The lock also supports a downgrade operation that converts an upgradable read lock into a regular read lock. We use it as follows: insertion proceeds as lookups but obtaining upgradable read locks. As soon as the thread determines that it will not need to update the node, it downgrades the lock before attempting to get an upgradable read lock on the next node down the path.

Deletion is more complex as it may remove a path that is being followed by other concurrent threads. CedrusDB assumes deletions are much less frequent than other operations. Based on this assumption, a thread performing a delete operation obtains write locks for the entire path, ruling out some concurrent access.

Fast Node Lock

The idiomatic way in Rust to add a lock to a type is `RwLock<T>`, a wrapper around type `T`. Because CedrusDB trie nodes are memory-mapped, we cannot

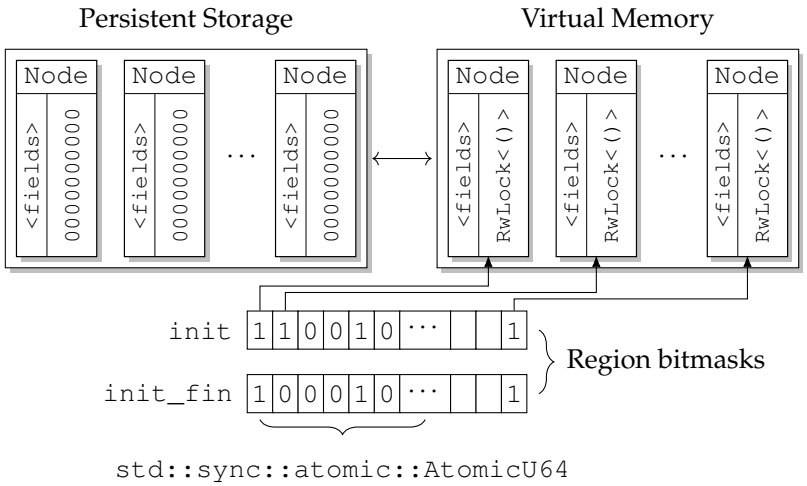


Figure 6.8: Implementation of trie node locks.

use this facility directly. Instead, we maintain a `RwLock<()>` within every trie node , using a neglibible amount of space in the node (which is dominated by the table of child pointers), but need a way to initialize the locks after mapping their space from disk.

The lock objects are initialized on-the-fly as threads try to obtain them. We maintain two bitmaps with each region, each with one bit per trie node. For a region size of 16MBytes full of 256-child tree nodes, less than 2KBytes is needed for the bitmasks. `init_fin` indicates whether the corresponding lock is initialized. `init` indicates whether some thread is in the process of initializing the lock. The bitmasks are implemented by a fixed-length array of 64-bit atomic variables so that query and modification can be performed atomically using bit operations (Figure 6.8). We hide this fast node lock facility from other parts of the system behind a safe interface.

Batched Writes

Like LevelDB, CedrusDB supports grouping several write operations (insert/delete) into a write batch. The atomicity of a write batch is guaranteed by write-ahead logging. Batches also need serializable isolation as they may be executed concurrently. Instead of acquiring a global mutex lock throughout the entire batch execution, CedrusDB implements a simple concurrency control method that divides a write batch execution into two phases. During the first phase, the global mutex is held, while the batch walks down the trie by alternating read locks and stops at the leaves where an insertion/deletion will happen. After the quick walk, the set of nodes whose write locks are required for all operations in the batch are recorded and deduplicated. Then the batch acquires the write locks and releases the global lock. With the node locks held, the thread resumes each operation in the batch and buffers all induced disk writes into a vector (WriteBatch in Figure 6.6). In addition to node locks, a thread may also acquire a lock to update the tail pointer of a logical space. Each tail pointer has its dedicated mutex lock. All tail pointer locks held by the thread will only be released at the end. Finally, the vector is submitted to the disk thread through the multi-producer, single-consumer write buffer, after which all locks are released. CedrusDB optimizes for single write operations without batching, as no extra concurrency control is needed in this case.

6.3 Evaluation

We evaluate the implementation of CedrusDB in order to answer the following questions:

- How does branching factor of a lazy-trie affect its performance? Which value of sluggishness should one choose? Is 256-bit practical enough for hashing? (Section 6.3.2, Section 6.3.2)
- What is the performance for various types of CedrusDB operations? Is it practical enough compared to other stores? (Section 6.3.2)
- How does performance degrade when user data cannot fit entirely in memory? What is the impact of the region size? (Section 6.3.2) What is the performance of data space allocator and what is the overall storage footprint? (Section 6.3.2)
- How well does CedrusDB perform under various kinds of realistic workloads? How well does it leverage concurrency? (Section 6.3.3)
- How does crash recovery in CedrusDB compare to other approaches? (Section 6.3.4)

6.3.1 Setup

Unless otherwise noted, we use Dell R340 servers to conduct our experiments. A server has a hexa-core Intel Xeon E-2176G 3.70GHz CPU with 64GB DDR4 memory. For the secondary storage medium, we use an Intel Optane 905P SSD with PCIe NVMe 3.0×4 interface. All evaluations are done on Ubuntu 18.04 LTS with a dedicated ext4 filesystem for persistent files. We allow asynchronous writes in all stores. For most experiments, the whole data set can fit into the memory.

We use FasterKV 1.8.4, LMDB 0.9.23 and RocksDB 6.1.2 as our baselines, representing a wide spectrum of persistent key-value store designs. In RocksDB,

we enable the additional cache feature by setting the LRU cache to 40GB, the same amount used for CedrusDB memory-mapped regions. We disabled data compression to make comparison more fair. LMDB requires specifying the maximum data store size upon creation to preallocate the storage space—we also set it to the same amount of memory.

Our main target for comparison is FASTER [47], as it is the key-value store that is most closely related to CedrusDB. Like CedrusDB, FASTER uses key hashes for indexing and thus does not support range queries or sorted iteration. Despite this similarity, FASTER has a very different persistence model. FASTER divides its memory into two parts. One section of memory serves as a data cache for fast access, while the other part, the *HybridLog*, is used to log updates. By default, FASTER only writes its HybridLog to disk when the log can no longer be held in memory in its entirety. But even when the log is flushed, a user still needs to manually invoke the checkpoint function to make the logged changes persistent. In practice, one has to decide how frequently one should invoke FASTER's checkpoint function, while other approaches persist their data continuously.

In our experiments, we made no FASTER checkpoints until the end of each run, resulting in the minimum I/O effort that FASTER undergoes to persist its state, but note that its in-memory index is not saved and is lost in the event of a failure. We made the initial hash table exactly large enough to contain the initial number of items and used the rest of the 40GB for HybridLog. To eliminate warmup-bias, we applied in-place updates until the memory part of the log was saturated and made a synchronous checkpoint to flush the leftover I/O induced by warmup operations before starting the workload. We used the C++

version of FASTER that is natively available on our Linux platform and should offer similar performance as the C# implementation [113]. FASTER also uses kernel-based AIO, so for fairness we set the same maximum I/O event limit as in CedrusDB.

For most microbenchmarks, we use 32-byte keys and 128-byte values. For macrobenchmarks, we evaluate CedrusDB using YCSB [50]. YCSB is written in Java whereas CedrusDB is in Rust and the other key-value stores are in C or C++. To avoid overhead caused by incompatible interfacing, we created a C binding for CedrusDB and invoke the APIs from a unified C++ test program dedicated to executing the YCSB workload. In the graphs with error bars, we run the same setup for 5 times and plot the mean in y-axis with standard deviation bars. For other graphs, the variation between runs is negligible. For each run, we populate the data store with a certain number of key-value items before running the test workload. The data store is reopened when the test starts. Finally, for multi-threaded experiments, we run a separate YCSB workload generator for each thread in such a way that a thread only handles keys that were preloaded or keys that were inserted by the thread itself.

6.3.2 Microbenchmarks

Branching Factor and Sluggishness

There are two parameters required to instantiate a lazy-trie: the tree branching factor and sluggishness, together determining the shape of the tree and its statistical characteristics. In Figure 6.9, we use branching factors 64, 128, and 256. We vary the sluggishness from 1 (no sluggishness) to 128. For each run, a

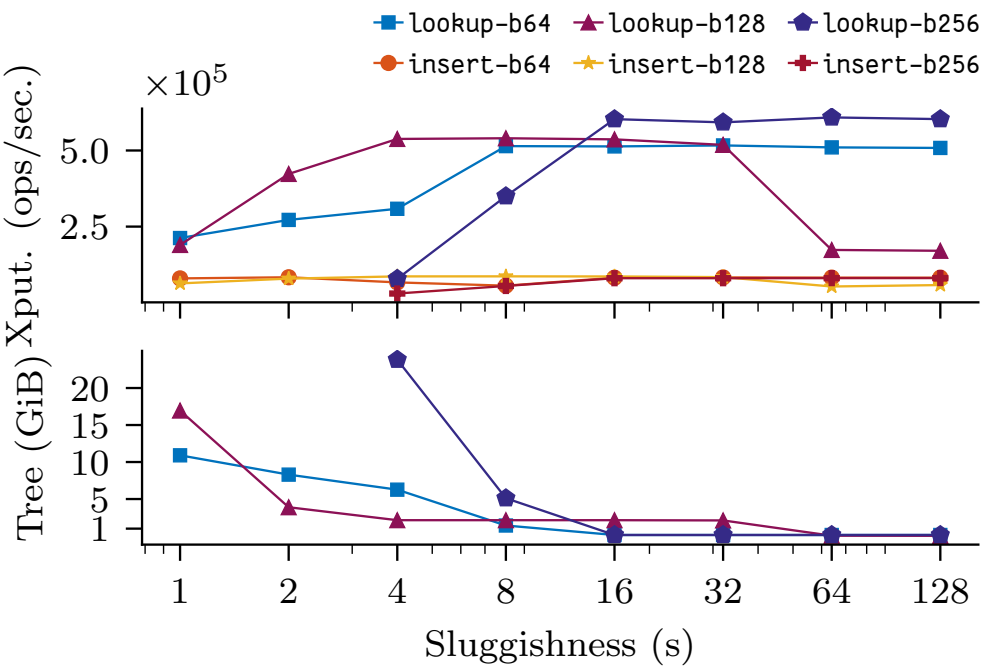


Figure 6.9: Lookup/insertion throughput (Xput.) and the tree footprint with different branching factors and sluggishness.

data store of 100 million items is used to perform 10 million uniformly random lookups or insertions.

As discussed in Section 6.1.3, we expect that a larger branching factor will make the efficacy of sluggishness more pronounced as having more slots in the child table leads to higher storage overhead and amplification. Indeed, we see that for a branching factor of 256, sluggishness significantly improves performance. Data points for $s < 4$ are not shown in the graphs because those runs end up with a tree footprint that exceeds the memory of our platform. Compared to (32+128)-byte key-values, each tree node with 256 children slots takes up around 2KBytes of space. By having more sluggishness, storage overhead is greatly mitigated and both lookup and insertion performance ramp up to reach or surpass those of other branching factors. When increasing sluggishness from 4 to 16, the memory footprint gets cut down from more than 23GBytes to

144MBytes. The performance drop of b128 with excessive sluggishness is due to the periodic change of node utilization (Figure 6.4, middle). Given our store size (10^8 items), this leads to excessively long linked lists. Thus one should choose the least sluggishness that achieves the desired footprint.

When using a branching factor of 64, each tree node only takes around 512 bytes and it requires a lower sluggishness for comparable performance. On the other hand, because the data store can be fully cached in memory, the choice of branching factor does not significantly affect the maximum performance due to the low cost of memory access. To reduce the tree height and have the best lookup performance, we use a branching factor of 256 with sluggishness of 16 as a practical choice for all subsequent experiments.

Hash Length

In the prior experiments, we used 256-bit HighwayHash to hash the keys. HighwayHash also provides 64- and 128-bit hashing. While generating a shorter hash is faster, it increases the collision rate, amplifying the cost of scanning the linked list of data nodes. We experimented with all three hash lengths but found no difference in performance. We use 256-bit hashes for the remainder of the evaluation.

Throughput

We examine the performance of individual types of operations with a single client thread. We populate the store with 100 million items and conduct 50 million reads or writes. Figure 6.10 shows that CedrusDB performs uniformly ran-

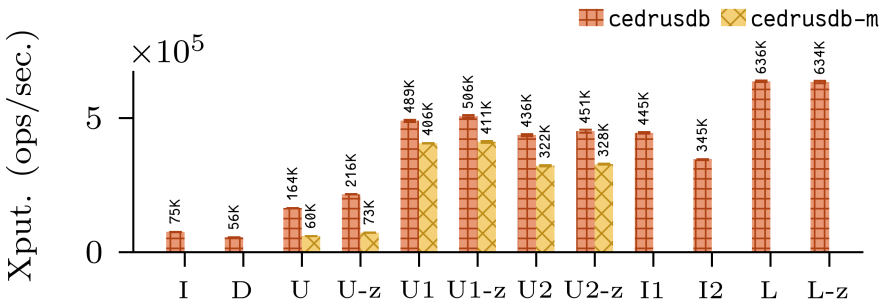


Figure 6.10: Microbenchmark with 10^8 items and 5×10^7 operations (single-threaded).

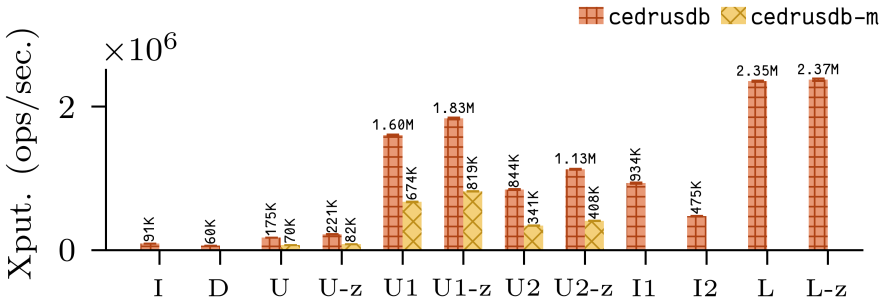


Figure 6.11: Concurrent microbenchmark with 10^8 items and 5×10^7 operations (4 threads).

dom (`L`) or Zipfian (`L-z`) lookups much faster than pure insertions (`I`), deletions (`D`) and updates (`U`). CedrusDB has two kinds of updates: (i) *in-place updates* (as used in FASTER/LMDB by default) when the new value can fit into the footprint of the current one, and (ii) *emulated updates* that remove the current values and insert the new ones. We instrumented CedrusDB so we could test both individually. `cedrusdb` shows in-place update performance and `cedrusdb-m` shows the performance when all updates are treated as deletions followed by insertions. Unsurprisingly, in-place updates are fastest since they do not alter the tree topology.

Although we used the fastest async channel implementation available for Rust, we still noticed insertions are bottlenecked by our write buffer as the

throughput of insertions would have been doubled (~180K) had we changed the order and granularity of writes (but atomicity would no longer be guaranteed). We observed the same bottleneck in Figure 6.9.

To see the impact of writes to the overall performance when they are mixed together with reads, we tested it with 10% (`U1*`, `I1`) and 20% (`U2*`, `I2`) writes. CedrusDB still benefits from fast reads in these workloads.

As for concurrent access (Figure 6.11), in a 4-thread setup, mixed writes also preserve some degree of scalability from reads. The performance of deletions is the lowest because they require frequent access to the data space allocator and also change the tree topology.

Elastic Memory and Regions

While CedrusDB is optimized for the case that the entire data store fits within a given memory budget, the region-based mapping design of CedrusDB allows a larger data store. We evaluate how performance degrades as CedrusDB runs out of the memory budget. We started with a baseline experiment using 100 million data items that did not have a memory budget, so the data store could utilize all the memory. We recorded the maximum number of regions and used that as the memory budget. We then experimented with 0%–25% of additional user data to see how performance changes.

Figure 6.12 shows lookup performance for region sizes ranging from 64KBytes (`lookup16`, 16-bit) to 16MBytes (`lookup24`, 24-bit). The figure that shows having larger region size results in better performance when all data fit in memory, but degrades quickly when data exceeds the memory budget. When

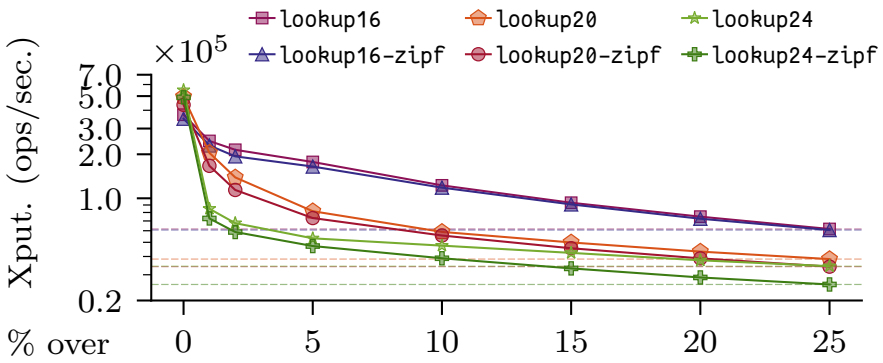


Figure 6.12: Read performance for different region sizes, with user data beyond the memory budget.

the whole data set fits in memory (0%), small region sizes hurt performance because there is more overhead for managing regions and their mapping. For large region sizes, the coarse granularity of mappings make it more likely that cold and hot items are collocated in the same region, resulting in increased swapping. Figure 6.13 shows the results for write operations. In this figure, each line is normalized to the full memory budget performance. We see that faster operations like updates degrade more than slower ones like deletions, due to the high penalty of swapping compared to in-memory operations. So while CedrusDB will continue to operate well when running out of the given memory budget rather than simply give up, it is important to adjust the memory budget accordingly.

Variable-Length Values and Fragmentation

So far (and as is common practice in many key-value store evaluations), we used the same length for all values. For CedrusDB, this means that the data space allocator only needs to take one step to find the next-fit location that was previously freed to recycle. When allowing in-place updates, the allocator is not

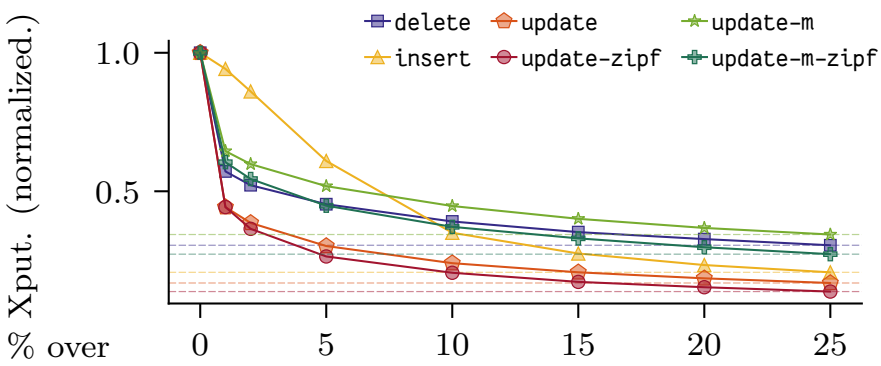


Figure 6.13: Normalized write performance with user data beyond the memory budget.

even engaged. Thus, to throughly examine our design and effectively evaluate the allocator, we generated a workload that mixes 128/256/1024 values uniformly. We first populated the store with 10 million items. To have each value updated many times on average, we ran 100 million operations with the mixed workload (`M`), changing the value of each key ~5 times throughout the entire run.

For the next-fit allocator, scanning through the entire free list before giving up is too expensive in practice. Instead of making sure to recycle a freed object whenever possible, allowing some slack in using the free list does not cause significant fragmentation. Therefore we limit the maximum number of steps in scanning the free list during an allocation.

In the right subgraph of Figure 6.14, we vary the scan step limit (`Max. Scan`) and show the change of the amplification factor (`Disk Amp.`) and the ratio of reusing freed space in an allocation (`Recycled`). The disk amplification factor is the final disk usage divided by the initial one. It is greater than 1 for all stores due to fragmentation. Once the limit exceeds 100 there is little difference compared to having no limit. Moreover, as shown in the

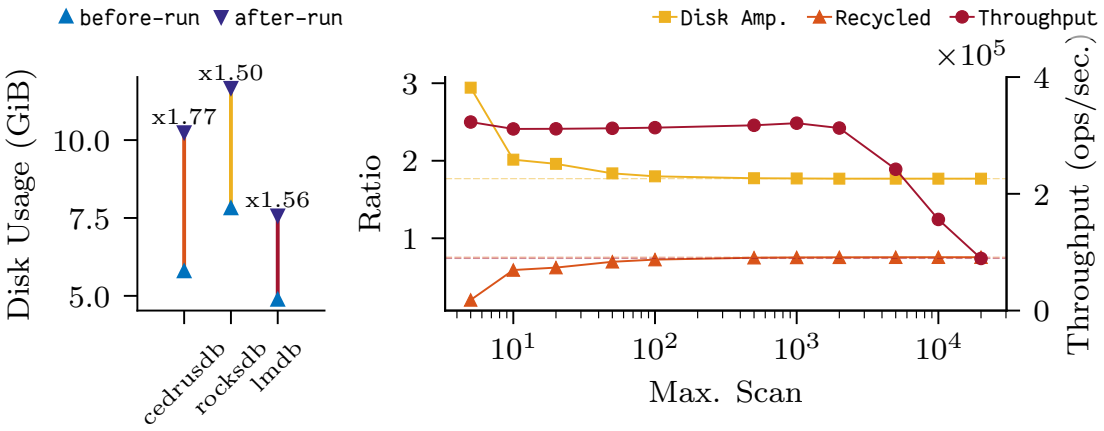


Figure 6.14: Data space allocator statistics and storage amplification due to fragmentation.

left subgraph of Figure 6.14, we believe the fragmentation ratio of CedrusDB is reasonable compared to other key-value stores.

6.3.3 Macrobenchmarks

In this section, we evaluate CedrusDB using YCSB [50] workloads with Zipfian distributions. We used a region size of 16 MBytes for all experiments.

Single-Threaded Performance

Figure 6.15 shows single-threaded throughput for different read/write ratios using 128-byte values; Figure 6.16 shows the same for 1KByte values. The data store is populated with 100 million keys. Two types of write operations are considered: updating the value of an existing key and inserting a value with a non-existing key. Although both use the same API, they trigger different code paths (as demonstrated in our microbenchmarks). The upper graph shows re-

sults from runs where all writes are updates whereas the lower shows insertions; in the rightmost bars, all operations are lookups. CedrusDB achieves performance comparable to or better than LMDB and RocksDB in write-intensive mixed workloads. It outperforms others in read-intensive cases, with the exception that LMDB is slightly faster in these cases and noticeably faster in pure lookups, when using 128-byte values.

In contrast, FASTER, which does not maintain an on-disk index, performs better with more updates. Changes are appended to an in-memory log that is eventually flushed to disk when memory runs low. In the absence of reads, this scheme is bottlenecked only by disk bandwidth. However, if there are read operations, FASTER performance suffers from not having an on-disk index. FASTER is optimized for in-place updates, which get translated into modifications inside the in-memory log. Its insertion performance reveals more of the performance impact by writes as shown at the bottom of the graph.

To evaluate performance when values have varying sizes, we generated two workloads. In workload A, for each key, the value is 128, 256 or 1024 bytes chosen uniformly at random. This illustrates a scenario in which the key-value store has items with only a few but very different sizes. On the other hand, workload B considers a scenario where most values have a similar size. The values are chosen between 128 and 256 bytes with Zipfian distribution. Both workloads are run using 100 million mixed lookup/update operations given 10 million initial items. CedrusDB achieves better performance than others for read-intensive cases in both workloads (Figure 6.18), while it degrades faster with more updates. This is because variable-length values may not be updated in-place and thus trigger intensive access to the data space allocator. On the

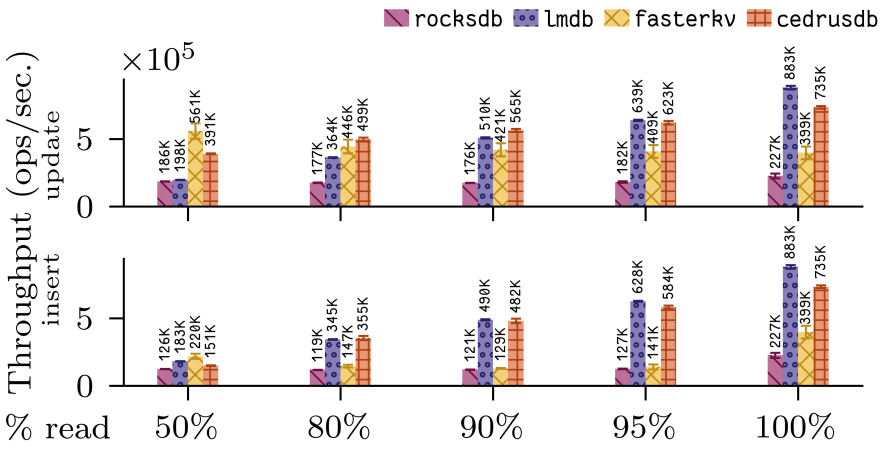


Figure 6.15: YCSB evaluation with 10^8 128-byte values and operations (single-threaded).

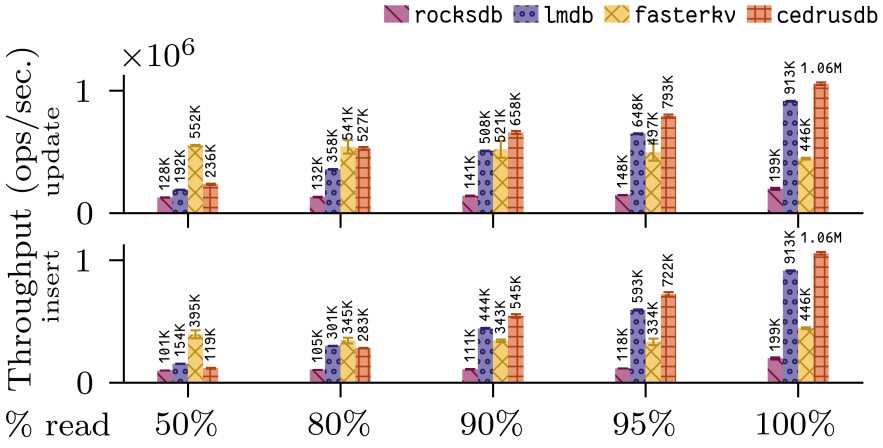


Figure 6.16: YCSB evaluation with 10^7 1024-byte values and operations (single-threaded).

other hand, even in the most write-intensive case the increase of disk usage caused by fragmentation of CedrusDB remains reasonable, as shown at the bottom of Figure 6.18. We did not evaluate the performance of FASTER for this workload as the C++ implementation (unlike its C# version) does not support variable-length values.

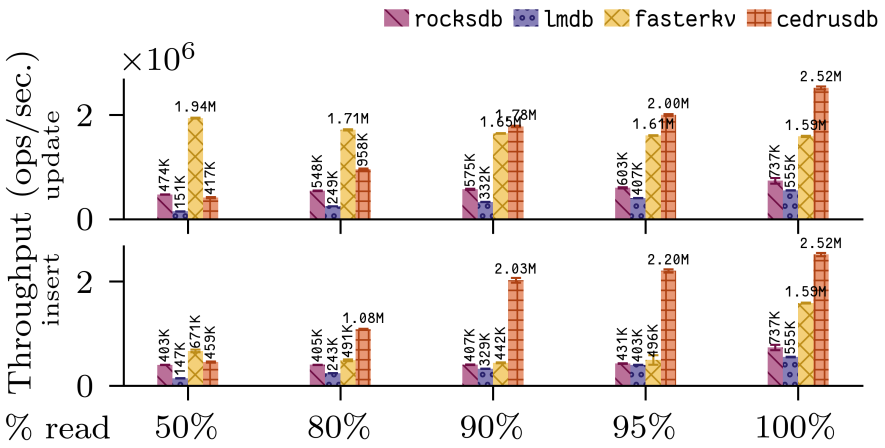


Figure 6.17: YCSB evaluation with 10^8 128-byte values and operations (4 threads).

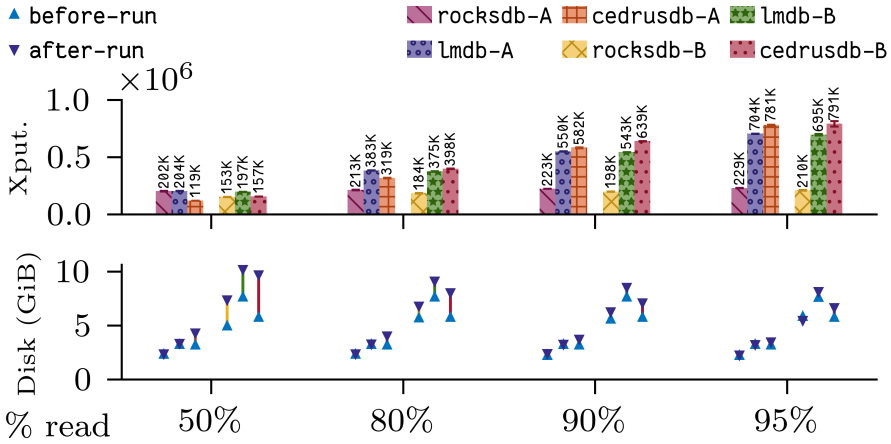


Figure 6.18: YCSB variable-length throughput (Xput.) with 128/256/1024 (`*-A`) or 128–256 (`*-B`) byte values (single-threaded).

Concurrent Performance

Next we evaluate multi-threaded performance. Both LMDB and RocksDB have specific optimizations to take advantage of concurrency [52, 74] whereas FASTER uses atomic operations on its in-memory hash table. Our lazy-trie requires no reorganization when keys are inserted or deleted, simplifying concurrent access, which could be viewed in a way as having small "hash tables" with

some tree hierarchy. Figure 6.17 shows the aggregated throughput when 4 client threads access the data store at the same time. CedrusDB outperforms others in read-intensive workloads. FASTER performs best under intensive updates, but the performance suffers when the writes are insertions.

To test the scalability in the number of threads, we conducted the same experiments on an Amazon AWS `r5d.8xlarge` instance. We used one NVMe SSD and run 80%–90% read and 20%–10% update/insert workloads with a varying number of client threads.[2] Figure 6.19 shows that CedrusDB scales well until it hits the shared lock bottleneck at around 8 threads. CedrusDB update performance plateaus much earlier than insertions because Zipfian updates induce contention of frequently accessed items (hashing does not prevent this because the same key always corresponds to the same hash), whereas insertions create new items and benefit more from non-overlapping tree walks. The Rust MPSC (Multi-Producer, Single-Consumer) write-buffer library limits insertion performance when there are more than 20 threads. Nevertheless, the read-intensive (90% read) update throughput of CedrusDB is still 1.05–3.07x that of RocksDB, 1.07–5.74x the throughput of LMDB, and 0.95–4.79x the throughput of FASTER. For insert, CedrusDB is either competitive or superior to all others. While the performance of FASTER is inferior to that of CedrusDB for various workloads, its in-memory, lock-free hash table implementation causes FASTER to plateau much later than CedrusDB in update workloads.

[2]FASTER experienced consistency issues when running with 24–28 threads, when some existing keys were reported missing. We went ahead and collected the measurements regardless.

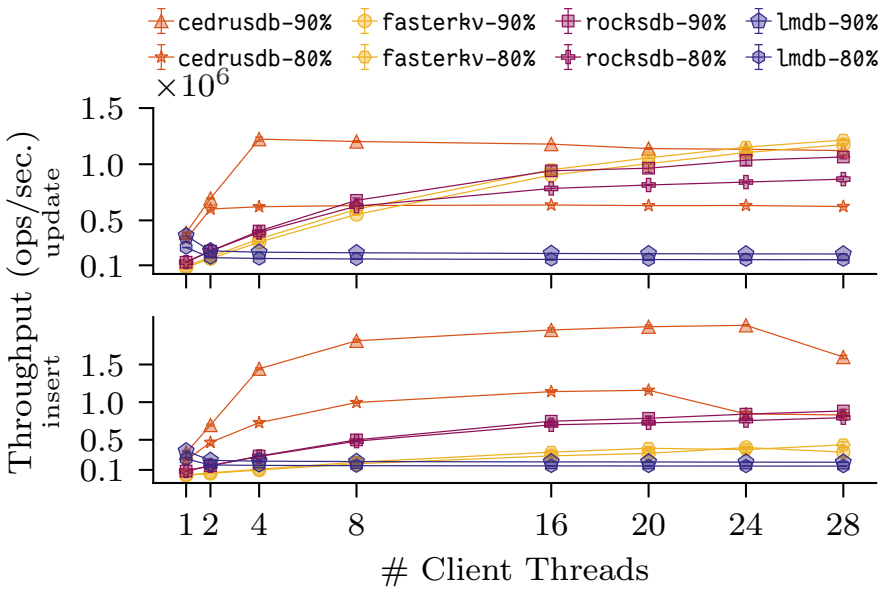


Figure 6.19: Concurrent YCSB evaluation with 10^8 128-byte values and operations (AWS NVMe SSD).

	Persistence	Indexing	Recovery (sec.)	Checkpoint (sec.)	Throughput (Kops/sec.)
CedrusDB	always, disk-indexed	hashed	0.645	-	391
FASTER	manual, log-based	hashed	28.16	19.38	446 (disk) 1125 (volatile)
LMDB	always, disk-indexed	sorted	0.003	-	198
RocksDB	always, disk-indexed	sorted	0.267	-	186
Masstree	always, log-based	sorted	31.8 per 10^8 ops	≈ 33.5*	1123
KVell	always, slab-based	sorted	91.9	-	410

Table 6.1: Key-value stores with different schemes. YCSB (50% updates), Zipfian, 128-byte values and 10^8 items. Throughput shows the normal operation performance where no checkpoint is made. * Extrapolated value—see text.

6.3.4 Crash Recovery

In recent years, various projects have proposed boosting the performance of key-value stores by eschewing the on-disk index and emphasizing a fast in-memory index. The potential price paid is increased recovery time. Similar to LMDB and RocksDB, CedrusDB takes a more traditional approach by hav-

ing an up-to-date on-disk index. In this section, we consider the performance trade-offs of the two approaches. We used CedrusDB, LMDB, and RocksDB as representatives of key-value stores that maintain an up-to-date on-disk index, and FASTER, Masstree [108], KVell [100] as representatives of the alternative approach.

We ran the following experiment with each key-value store. After populating the store with 100 million keys, we ran 100 million 50% update YCSB workload operations before killing the program. We then measured the recovery time.

As shown in Table 6.1, the sorted on-disk index of RocksDB comes at the expense of throughput compared to CedrusDB whose index with hashed keys does not support range queries. RocksDB has faster recovery time because its WAL records high-level operations, whereas CedrusDB logs the induced low-level writes. Nonetheless, the recovery time for both is short and mostly dependent on the implementation and configuration and not affected by store size or total number of writes. LMDB uses *shadow paging*, a copy-on-write technique to persist data. Without the need for WAL, it takes neglibible time to recover. Next we compare with FASTER, Masstree and KVell.

FASTER. As recommended by the developers [114], we made checkpoints with the "fold-over" setup, which result in lightweight, incremental checkpoints. Each fold-over checkpoint blocked on-going operations for around 19 seconds, and recovery took 28 seconds. The overhead of the checkpoint is partially due to saving the in-memory hash table, a limitation of using flat structure that is not friendly to persistent storage. For its normal performance, we used

the same hash function as in CedrusDB and also tried the "volatile" case where there is no warm-up phase to saturate the memory part of HybridLog so there are no disk writes. In this case, we found the performance similar to Masstree.

Masstree. Masstree [108] maintains a trie-like concatenation of B^+-trees in memory and logs all writes to the disk. This enables high throughput as no on-disk index is maintained. On the other hand, during a recovery the log has to be replayed to restore the in-memory index. In the experiment we performed, it took around 31.8 seconds for playing back around 50 million write operations (50% of operations are writes).

To mitigate this and also to recycle the log storage, Masstree supports checkpoints that serve as new initial states for replay. We took snapshots of the store at 1 million (0.29 seconds) to 10 million (3.35 seconds) items and confirmed linear growth of the checkpointing time. Unfortunately, the snapshot with 100 million items was not successful with the Masstree code, so the number in Table 6.1 is extrapolated.

KVell. Instead of using an append-only log, KVell [100] uses slabs to preallocate space for items. We used its most recent GitHub code to run the experiment. The code currently does not support YCSB keys (the paper used 8-byte random integer keys for YCSB instead). We generated 23-byte keys with random bytes using the same distribution to best approximate the YCSB workload. KVell's recovery takes 91.9 seconds regardless of the number of operations. This time depends only on the size of the data store. KVell is currently unable to perform a crash recovery for a workload with a mix of insert and delete operations.

6.4 Related Work

Various prior systems have looked into better leveraging available memory to speed up performance of key-value stores. SILT [103] has a pipeline of three data structures to improve memory-efficiency and write-friendliness. LMDB [53] is a popular open-source key-value store that leverages a memory-mapped, copy-on-write B^+-tree. CedrusDB's usage of memory-mapped storage is inspired by LMDB. As discussed in Section 6.3.4, there are also persistent key-value stores that only keep a fast, concurrent hash table or other indexing data structures in-memory and pipe all writes directly into append-only logs or pre-allocated slabs [47, 100, 108, 111]. Such architectures suffer from significant recovery/checkpoint overhead.

There has been extensive work to reduce write-amplification in LSM-based persistent key-value stores. RocksDB [42, 66] is a fork of LevelDB improved by developers at Facebook. It provides more features such as multi-threaded compaction and support for transactions. Inspired by skip lists and based on HyperLevelDB [71], PebblesDB [130] proposes the Fragmented LSM data structure, carefully choosing the SSTs during compaction to reduce amplification. LSM-trie [153] uses a static hash-trie merge structure that keeps reorganizing data for more efficient compaction. SuRF [157] uses an LSM design with a trie-based filter to optimize range queries. Accordion [32] improves the memory organization for LSM. Monkey [58] reduces the lookup cost for LSM by allocating memory to filter across different levels, minimizing the number of false positives. Dostoevsky [59] introduces lazy-leveling to remove superfluous merging. mLSM [131] is tailored for blockchain applications and significantly improves the performance of the Ethereum storage subsystem.

There are also recent proposals to combine LSM and B$^+$-tree designs. Jungle [12] reduces update cost without sacrificing lookup cost in LSM using a B$^+$-tree. SLM-DB [85] assumes persistent memory hardware. It uses a B$^+$-tree for indexing and stages insertions to LSM.

Existing storage data structures have evolved in response to changes in hardware. wB$^+$-tree [48] reduces transaction logging overhead for a B$^+$-tree in non-volatile main memory. LB$^+$-tree [105] optimizes the index performance using 3DXPoint persistent memory. S3 [158] uses an in-memory skip-list index for a customized version of RocksDB in Alibaba Cloud. RECIPE [98] offers a principled way to convert concurrent indexes on DRAM to the one on persistent-memory with crash-consistency.

Like CedrusDB, other systems have embraced the trie for in-memory indexes. ART [99] uses a radix tree, also compresses the non-diverging paths. HOT [27] uses an adaptive number of children for each node. Compared to these in-memory indexes, there are several major differences: (1) CedrusDB is designed to be persistent and has an optimized lazy-trie for an on-disk index; (2) To ensure near-optimal tree height like a B$^+$-tree, instead of directly using key strings to index the trie, lazy-trie uses fixed-length hashes, having different statistical properties; (3) In addition to path compression, lazy-trie employs sluggish splitting to reduce variance in tree height, making the tree storage footprint practical and even lower than that for a B$^+$-tree.

CHAPTER 7

CONCLUSION

The dissertation has explored the design space of practical Byzantine fault tolerant replication (consensus) protocols and persistent key-value stores, both of which are at the core of blockchain infrastructures.

HotStuff revolves around a three-phase core, allowing a new leader to simply pick the highest QC it knows of. This alleviates the complexity in traditional protocols and at the same time considerably simplifies the leader replacement protocol. Having (almost) canonized the phases, it is easy to pipeline HotStuff and to frequently rotate leaders.

Taking a bolder approach, we sketched the Snow protocols. These protocols are efficient and robust, combining some of the best features from classical and Nakamoto consensus. They scale well, achieve high throughput and quick finality, work without precise membership knowledge, and degrade gracefully under catastrophic adversarial attacks. We also built a decentralized payment system prototype called Avalanche. There is much work to do to continue this line of research. One improvement could be the introduction of an adversarial network scheduler. Another improvement would be to characterize the system's guarantees under an adversary whose powers are realistically limited, whereupon performance would improve even further. More sophisticated initialization mechanisms would improve liveness of multi-value consensus.

Finally, we explored the realm of storage systems and in particular the idea of an in-memory index that is also storage-friendly, allowing both fast in-memory access and fast failure recovery. We designed the lazy-trie data struc-

ture and implemented CedrusDB to this end. CedrusDB represents a new trade-off between fast access and fast recovery. Potential future work directions include further optimizing performance, for example by removing bottlenecks in shared locks and the Rust write buffer. Furthermore, to increase applicability, we are also interested in designing a sorted data structure that could support range queries.

7.1 On Designing New Protocols

From our experience with BFT replication protocols, we realized the importance of decoupling safety from liveness. Protocols in the past tend to blend these two aspects of a system together. For example, PBFT's view change sub-protocol states not only how to properly carry on the accepted state change across failures, but also the mechanisms to eventually always make progress. This includes the use of timeouts and leader election heuristics, where some concrete solution is assumed to exist, without a detailed, well-defined specification. According to our interaction with the industry, the agreement part of a replication protocol is already subtle and difficult to implement, not to mention the combination of both. For engineers, they would like a protocol that has a relatively small code base to improve its correctness, while they can fearlessly tweak and customize their own heuristics for its liveness and efficiency. We believe both HotStuff and Snowball can offer such separation and help accelerate the industrialization of BFT replication.

Another common deficiency, particularly in proposals for blockchain designs, is a clear specification of the core protocol (i.e., a distributed algorithm).

Many works tend to describe a protocol in plain English, in a step-by-step manner. This is often misleading and confusing to those who try to learn or implement the protocol, or researchers who would like to analyze or verify the protocol. In a protocol that deals with asynchrony, it is better to assume that different types of messages can arrive at any time or in any order of the protocol execution and their handling may interfere with each other by changing the execution state. Many prior works contain the pseudocode that sequentially lay out the handling logic, entangled with the notion of time, rounds or different phases of the protocol, which lacks a clear definition. Or they sometimes mix the description of sequential and concurrent logic. In both of our final protocols, we deliberately took an "event-driven" or transitional style of pseudocode in the specification. We adopt a style similar to the one used in a classic textbook [39] about distributed programming. We generally assume the sequential logic within each "handler" of an asynchronous event is atomic so that it cannot be interrupted by other external events, whereas the events may trigger different handlers in an arbitrary order. It makes it easy and straightforward to implement the proposed protocol within a modern programming environment. Although in theory, it is always possible to formally specify a protocol with something like TLA$^+$ [96], researchers rarely do that because the specification, while being friendly to provers and model checkers, is difficult to extract the operational logic from. Specifying a full fledged complex protocol may take substantial amount of time and the final product may be lengthy. Thus, writing the pseucocode for a protocol is still in demand, and becomes an art of balancing readability and formality.

7.2 On Reducing Cost and the Leader Bottleneck

One main research interest in BFT replication is to improve its efficiency for practical implementations, which was once the major obscable that hindered its industrial adoption. In this dissertation, we managed to achieve good performance in both HotStuff and the Snow family, under their respective practical models.

In HotStuff, our evaluation code took a conservative approach that uses a concatenated list of individual signatures to implement QCs. One of the reasons for why we did it without actual threshold signatures is that known solutions were not yet in a usable state. There was no robust, fast and ready-to-use library for something like BLS [31]. The actual verification or generation cost for a combined signature might even be more expensive in practice when the number of replicas is small, compared to having the list of single signatures. Thus, although theoretically HotStuff is at the lower bound of the complexity, this advantage is perhaps more pronounced only after the underlying cryptography gets more efficient, as it always does.

HotStuff does not address the leader bottleneck issue. The bottleneck, as explained and discussed at the end of Section 2.3.3, is inherently caused by two factors. One is the strong consistency requirement by SMR, that the same, linear log has to be agreed upon by all correct replicas, which inevitably introduces contention as for whose proposal should be agreed upon. Some existing works may argue they have "removed" the leader bottleneck because they do not even have the notion for this special role in the protocol, but this totally democratic proposal actually introduces more contention to the agreement, so some extra

measure is required to make sure the preferred proposal is unique or likely to be unique (such as using some cryptographic primitive to help converge, after a number of stochastic, repeated rounds, which incurs more overhead). There are also works that use additional tricks that are in effect batching, not addressing the fundamental issue. For this factor, our opinion is that it is a natural consequence from totally ordered replication, and the only way to get around it is to assume some weaker replication model, as in EPaxos [117], which, of course, makes it more ad-hoc to the application or service it supports. The leadership in partially synchronous protocols migitates the situation by introducing a hierarchy, just like a democratic government may still have a senate, congress or parliament, together with a president or prime minister as the leader. This is more efficient than letting every single person propose and vote in legislation. The real-world politics also requires agreement with unknown adversaries, and the analogy may be interesting and inspiring.

The other factor is the basic assumption that at least some node needs to exchange messages to the vast majority of others in order to make a decision. This sounds fair because if nodes do not talk comprehensively, some nodes may not effectively participate in the consensus at all. As the other major source of the bottleneck, it implies at least one node has to communicate with $O(n)$ nodes in the system. While the actual bandwidth utilization could be still low, thanks to the small footprint of messages, the processing cost (i.e., to react to messages) is typically not scalable. Unlike distributed (parallel) computing algorithms for function optimization or machine learning, consensus protocols have a more or less sequential logic at their core. Thus, given the limited capacity of sequential processing, such growth still becomes the major bottleneck. For example, Google typically keeps 5 machines for its Paxos replication group, as it is un-

likely to simultaneously encounter more than two failing replicas, and having more replicas degrades the performance superlinearly.

Our second work in this dissertation, however, gave some hints on how to circumvent both factors of the leader bottleneck. The Snowball protocol does not require an $O(n)$ for information exchange in each round, and there are only a logarithmic number of rounds. This is definitely not magic nor a free lunch, but a result from stochastic sampling and approximation. The protocol suggests it is both possible and sensible to have probabilistic safety that is stronger than Nakamoto consensus in a practical blockchain system. Furthermore, Avalanche also showed it is not necessary to have a standard State Machine Replication structure in order to support a payment system, as transactions do not need to be fully ordered. The scalability of Avalanche partly comes from avoiding the aforementioned contention in linear replication.

7.3 "..., Long Live the Consensus!" — What Would be Next?

Even though the blockchain development reignited research interest in BFT consensus, we find that this field is converging. While it may sound like bad news for consensus research, because it becomes harder to come up with new improvements or contributions, it is a healthy sign for industry. It is unimaginable to see white papers about consensus protocols (even those incorrect ones) by blockchain projects flying around for years, without having an implementation consolidated faithfully from some of them for an actual platform. The dream of having a decentralized platform for finance and computation goes on and there is just so much to do. After more and more reliable and practical implementa-

tions of BFT replication go online, the next focus will likely move up to the replicated state machine level. Admittedly, in the past, the state machine itself is not a very interesting topic any more, as it is dependent upon the applications that the system supports. However, unlike centralized service providers who can always roll out major changes to the infrastructure hidden from their customers, a decentralized service provider needs a more general approach. To enable a general-purpose service like Ethereum, the state machine itself has some necessary abstraction (its virtual ISA, for example) and thus becomes non-trivial to optimize. We guess that blockchain storage, especially "verifiable (or authenticated) storage," is going to be a major bottleneck and candidate for advances in performance. This is not limited to the underlying embedded key-value store, but may also include the interaction to the state machine execution or even the consensus (e.g., state transfer, snapshots, etc.) At last, consensus is not "dead," but just getting more mature, and while it is always good to start solving a problem by isolation, there is also time to think more globally about where it fits in the whole picture of a blockchain infrastructure and what it takes to keep pushing the performance of the entire system.

The exploration of performance scaling never ends.

APPENDIX A

PROOF OF SAFETY FOR CHAINED HOTSTUFF

Theorem 5. *Let b and w be two conflicting nodes. Then they cannot both become committed, each by an honest replica.*

Proof. We prove this theorem by contradiction. By an argument similar to Lemma 1, b and w must be in different views. Assume during an exectuion b becomes committed at some honest replica via the QC Three-Chain b, b', b'', b^*, likewise, w becomes committed at some honest replica via the QC Three-Chain w, w', w'', w^*. Since each of b, b', b'', w, w', w'' get its QC, then w.l.o.g., we assume b is created in a view higher than w'', namely, $b'.justify.viewNumber > w^*.justify.viewNumber$, as shown in Figure A.1.

We now denote by v_s the lowest view higher than $v_{w''} = w^*.justify.viewNumber$ in which there is a qc_s such that $qc_s.node$ conflicts with w. Let $v_b = b'.justify.viewNumber$. Formally, we define the following predicate for any qc:

$$E(qc) := (v_{w''} < qc.viewNumber \leq v_b) \wedge (qc.node \text{ conflicts with } w).$$

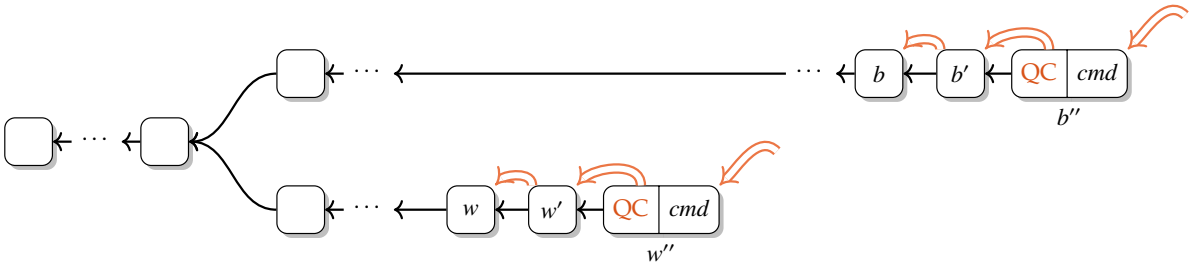


Figure A.1: w and b both getting committed (impossible). Nodes horizontally arranged by view numbers.

We can now set the *first* switching point qc_s:

$$qc_s := \arg\min_{qc}\{qc.viewNumber \mid qc \text{ is valid} \wedge E(qc)\}.$$

By assumption, such qc_s exists, for example, qc_s could be $b'.justify$. Let r denote a correct replica in the intersection of $w^*.justify$ and qc_s. By assumption on the minimality of qc_s, the lock that r has on w is not changed before qc_s is formed. Now consider the invocation of SAFENODE in view v_s by r, with a message m carrying a conflicting node $m.node = qc_s.node$. By assumption, the condition on the lock (Line 26 in Algorithm 1) is false. On the other hand, the protocol requires $t = m.node.justify.node$ to be an ancestor of $qc_s.node$. By minimality of qc_s, $m.node.justify.viewNumber \leq v_{w''}$. Since $qc_s.node$ conflicts with w, t cannot be w, w' or w''. Then, $m.node.justify.viewNumber < w'.justify.viewNumber$, so the other half of the disjunct is also false. Therefore, r will not vote for $qc_s.node$, contradicting the assumption of r.

□

The liveness argument is almost identical to Basic HotStuff, except that we have to assume after GST, two consecutive leaders are correct, to guarantee a decision. It is omitted for brevity.

APPENDIX B

PROOF OF SAFETY FOR HOTSTUFF IMPLEMENTATION PSEUDOCODE

Lemma 6. *Let b and w be two conflicting nodes such that $b.height = w.height$, then they cannot both have valid quorum certificates.*

Proof. Suppose they can, so both b and w receive $2f + 1$ votes, among which there are at least $f + 1$ honest replicas voting for each node, then there must be an honest replica that votes for both, which is impossible because b and w are of the same height. □

Notation 1. For any node b, let "$\leftarrow$" denote parent relation, i.e. $b.parent \leftarrow b$. Let "$\overset{*}{\leftarrow}$" denote ancestry, that is, the reflexive transitive closure of the parent relation. Then two nodes b, w are conflicting iff. $b \not\overset{*}{\leftarrow} w \wedge w \not\overset{*}{\leftarrow} b$. Let "$\Leftarrow$" denote the node a QC refers to, i.e. $b.justify.node \Leftarrow b$.

Lemma 7. *Let b and w be two conflicting nodes. Then they cannot both become committed, each by an honest replica.*

Proof. We prove this important lemma by contradiction. Let b and w be two conflicting nodes at different heights. Assume during an execution, b becomes committed at some honest replica via the QC Three-Chain $b(\Leftarrow \wedge \leftarrow)b'(\Leftarrow \wedge \leftarrow)b'' \Leftarrow b^*$; likewise, w becomes committed at some honest replica via the QC Three-Chain $w(\Leftarrow \wedge \leftarrow)w'(\Leftarrow \wedge \leftarrow)w'' \Leftarrow w^*$. By Lemma 1, since each of the nodes b, b', b'', w, w', w'' have QC's, then w.l.o.g., we assume $b.height > w''.height$, as shown in Figure A.1.

We now denote by qc_s the QC for a node with the lowest height larger than $w''.height$, that conflicts with w. Formally, we define the following predicate for

any qc:

$$E(qc) \coloneqq (w''.height < qc.node.height \leq b.height) \wedge (qc.node \text{ conflicts with } w)$$

We can now set the first switching point qc_s:

$$qc_s \coloneqq \arg\min_{qc}\{qc.node.height \mid qc \text{ is valid} \wedge E(qc)\}.$$

By assumption, such qc_s exists, for example, qc_s could be $b'.justify$. Let r denote a correct replica in the intersection of $w^*.justify$ and qc_s. By assumption of minimality of qc_s, the lock that r has on w is not changed before qc_s is formed. Now consider the invocation of ONRECEIVEPROPOSAL, with a message carrying a conflicting node b_{new} such that $b_{new} = qc_s.node$. By assumption, the condition on the lock (Line 17 in Algorithm 4) is false. On the other hand, the protocol requires $t = b_{new}.justify.node$ to be an ancestor of b_{new}. By minimality of qc_s, $t.height \leq w''.height$. Since $qc_s.node$ conflicts with w, t cannot be w, w' or w''. Then, $t.height < w.height$, so the other half of the disjunct is also false. Therefore, r will not vote for b_{new}, contradicting the assumption of r. □

Theorem 8. *Let cmd_1 and cmd_2 be any two commands where cmd_1 is executed before cmd_2 by some honest replica, then any honest replica that executes cmd_2 must executes cmd_1 before cmd_2.*

Proof. Denote by w the node that carries cmd_1, b carries cmd_2. From Lemma 6, it is clear the committed nodes are at distinct heights. Without loss of generality, assume $w.height < b.height$. The commit of w are b are triggered by some ONCOMMIT(w') and ONCOMMIT(b') in UPDATE, where $w \overset{*}{\leftarrow} w'$ and $b \overset{*}{\leftarrow} b'$. According to Lemma 7, w' must not conflict with b', so w does not conflict with b. Then $w \overset{*}{\leftarrow} b$, and when any honest replica executes b, it must first executes w by the recursive logic in ONCOMMIT. □

B.1 Remarks

In order to shed insight on the tradeoffs taken in the HotStuff design, we explain why certain constraints are necessary for safety.

Why monotonic vheight? Suppose we change the voting rule such that a replica does not need to vote monotonically, as long as it does not vote more than once for each height. The weakened constraint will break safety. For example, a replica can first vote for b and then w. Before learning about b', b'', it first delivers w', w'', assuming the lock is on w, and vote for w''. When it eventually delivers b'', it will flip to the branch led by b because it is eligible for locking, and b is higher than w. Finally, the replica will also vote for b'', causing the commit of both w and b.

Why direct parent? The direct parent constraint is used to ensure the equality $b.height > w''.height$ used in the proof, with the help of Lemma 6. Suppose we do not enforce the rule for commit, so the commit constraint is weakened to $w \overset{*}{\leftarrow} w' \overset{*}{\leftarrow} w''$ instead of $w \leftarrow w' \leftarrow w''$ (same for b). Consider the case where $w'.height < b.height < b'.height < w''.height < b''.height$. Chances are, a replica can first vote for w'', and then discover b'' to switch to the branch by b, but it is too late since w could be committed.

BIBLIOGRAPHY

[1] madvise(2) - Linux manual page. http://man7.org/linux/man-pages/man2/madvise.2.html. Accessed: 2020-04-15.

[2] Casper FFG with one message type, and simpler fork choice rule. https://ethresear.ch/t/casper-ffg-with-one-message-type-and-simpler-fork-choice-rule/103, 2017.

[3] Istanbul BFT's design cannot successfully tolerate fail-stop failures. https://github.com/jpmorganchase/quorum/issues/305, 2018.

[4] A livelock bug in the presence of byzantine validator. https://github.com/tendermint/tendermint/issues/1047, 2018.

[5] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[6] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[7] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance: Thelma, Velma, and Zelma. *CoRR*, abs/1801.10022, 2018.

[8] Ittai Abraham and Dahlia Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 123, 2017.

[9] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, pages 25:1–25:19, 2017.

[10] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Report 2019/270, 2019. https://eprint.iacr.org/2019/270.

[11] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29-August 2, 2019*, 2019.

[12] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining LSM-tree and copy-on-write B+-tree. In Daniel Peek and Gala Yadgar, editors, *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. USENIX Association, 2019.

[13] Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. Fast keyed hash/pseudo-random function using SIMD multiply and permute, 2016.

[14] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

[15] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo web browser engine using rust. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 81–89. ACM, 2016.

[16] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[17] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged byzantine impostors. Technical report, Technical Report YALEU/DCS/TR-1332, Yale University Department of Computer Science, 2005.

[18] Hagit Attiya, Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122–152, 1994.

[19] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, 2015.

[20] Adam Back. Hashcash - a denial of service counter-measure. 2002.

[21] Leemon Baird. Hashgraph consensus: fair, fast, byzantine fault tolerance. Technical report, Swirlds Tech Report, 2016.

[22] Siddhartha Banerjee, Avhishek Chatterjee, and Sanjay Shakkottai. Epidemic thresholds with external agents. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2202–2210. IEEE, 2014.

[23] Dave Bayer, Stuart Haber, and W Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences Ii*, pages 329–334. Springer, 1993.

[24] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30, 1983.

[25] Iddo Bentov, Pavel Hubácek, Tal Moran, and Asaf Nadler. Tortoise and Hares Consensus: the Meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017:300, 2017.

[26] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 355–362, 2014.

[27] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A height optimized trie index for main-memory database systems. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM, 2018.

[28] Bitnodes. Global Bitcoin nodes distribution. `https://bitnodes.earn.com/`. Accessed: 2018-04.

[29] blockchain.com. Median confirmation time. `https://www.blockchain.com/charts/median-confirmation-time`. Accessed: 2021-06-12.

[30] blockchain.com. Transaction rate per second. https://www.blockchain.com/charts/transactions-per-second. Accessed: 2021-06-12.

[31] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil Pairing. *J. Cryptology*, 17(4):297–319, 2004.

[32] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better memory organization for LSM key-value stores. *PVLDB*, 11(12):1863–1875, 2018.

[33] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[34] Gabriel Bracha. An $O(\log n)$ expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

[35] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, 2016.

[36] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[37] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350. USENIX Association, 2006.

[38] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

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

[40] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 18(3):219–246, 2005.

[41] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[42] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[43] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186, 1999.

[44] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[45] Central Intelligence Agency. The world factbook. `https://www.cia.gov/the-world-factbook/countries/denmark/#energy`. Accessed: 2021-03.

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

[47] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 275–290. ACM, 2018.

[48] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[49] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 277–290, 2009.

[50] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[51] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[52] Symas Corporation. LMDB: Lightning Memory-Mapped Database Manager. `http://www.lmdb.tech/doc/`. Accessed: 2020-05-16.

[53] Symas Corporation. Symas Lightning memory-mapped database. `https://symas.com/lmdb/`. Accessed: 2020-04-15.

[54] 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*, pages 177–190. USENIX Association, 2006.

[55] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (a position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.

[56] Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `https://eprint.iacr.org/2016/919`.

[57] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 66–98, 2018.

[58] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In Semih Salihoglu, Wenchao Zhou, Rada

Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 79–94. ACM, 2017.

[59] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 505–520. ACM, 2018.

[60] 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. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

[61] Redox OS Developers. Redox - your next(gen) OS. `https://www.redox-os.org/`. Accessed: 2020-04-15.

[62] The Rust Project Developers. The Rustonomicon: the dark arts of advanced and unsafe Rust programming. `https://doc.rust-lang.org/stable/nomicon/`. Accessed: 2020-04-15.

[63] Digiconomist. Bitcoin energy consumption index. `https://digiconomist.net/bitcoin-energy-consumption`. Accessed: 2021-03.

[64] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

[65] Danny Dolev and H. Raymond Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 401–407, 1982.

[66] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in RocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[67] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[68] Moez Draief, Ayalvadi Ganesh, and Laurent Massoulié. Thresholds for virus spread on networks. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools*, page 51. ACM, 2006.

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

[70] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.

[71] Robert Escriva. Inside HyperLevelDB. `https://hackingdistributed.com/2013/06/17/hyperleveldb/`. Accessed: 2020-04-20.

[72] Etherscan. Ethereum average block time chart. https://etherscan.io/chart/blocktime. Accessed: 2021-06-12.

[73] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59, 2016.

[74] Facebook. Memtable - facebook/rocksdb wiki. `https://github.com/facebook/rocksdb/wiki/MemTable#concurrent-insert`. Accessed: 2020-05-16.

[75] Hal Finney. Rpow-reusable proofs of work. *Internet: https://cryptome.org/rpow.htm*, 2004.

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

[77] Ayalvadi Ganesh, Laurent Massoulié, and Don Towsley. The effect of network topology on the spread of epidemics. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1455–1466. IEEE, 2005.

[78] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin Backbone Protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

[79] 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, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

[80] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

[81] Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly dynamic distributed computing with byzantine failures. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 176–183. ACM, 2013.

[82] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[83] Insider. Alipay overtakes PayPal as the largest mobile payments platform in the world. https://www.businessinsider.com/alipay-overtakes-paypal-as-the-largest-mobile-payments-platform-in-the-world-2014-2. Accessed: 2021-06-12.

[84] Mark S Johnstone and Paul R Wilson. The memory fragmentation problem: Solved? *ACM Sigplan Notices*, 34(3):26–36, 1998.

[85] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 191–205. USENIX Association, 2019.

[86] 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*, pages 295–308. ACM, 2012.

[87] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, 2009.

[88] Matt J Keeling and Pejman Rohani. *Modeling infectious diseases in humans and animals*. Princeton University Press, 2011.

[89] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

[90] Donald Ervin Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, volume 1. Pearson Education, 1997.

[91] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 279–296, 2016.

[92] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[93] Chinmay Kulkarn, Badrish Chandramouli, and Ryan Stutsman. Achieving high throughput and elasticity in a larger-than-memory store. In *Proc. VLDB Endow. Volume 14, Issue 8, 2021 (to appear)*.

[94] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[95] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[96] Leslie Lamport. Specifying concurrent systems with TLA$^+$. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:183–250, 1999.

[97] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[98] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 462–477. ACM, 2019.

[99] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[100] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 447–461. ACM, 2019.

[101] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling Nakamoto consensus to thousands of transactions per second. *CoRR*, abs/1805.03870, 2018.

[102] Thomas M. Liggett. Stochastic models of interacting systems. *The Annals of Probability*, 25(1):1–29, 1997.

[103] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 1–13. ACM, 2011.

[104] Microsoft LinkedIn. Project Voldemort - a distributed database. `https://www.project-voldemort.com/voldemort`. Accessed: 2020-12-10.

[105] Jihang Liu, Shimin Chen, and Lujun Wang. LB+-trees: Optimizing persistent index performance on 3D XPoint memory. *PVLDB*, 13(7):1078–1090, 2020.

[106] OrientDB Ltd. OrientDB. `https://www.orientdb.org/`. Accessed: 2020-12-10.

[107] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[108] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM, 2012.

[109] Leonardo Mármol, Jorge Guerra, and Marcos K. Aguilera. Non-volatile memory through customized key-value stores. In Nitin Agrawal and Sam H. Noh, editors, *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[110] David Mazieres. The Stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 2015.

[111] Alexander Merritt, Ada Gavrilovska, Yuan Chen, and Dejan S. Milojicic. Concurrent log-structured memory for many-core key-value stores. *PVLDB*, 11(4):458–471, 2017.

[112] James Mickens. The saddest moment. *;login:*, 39(3), 2014.

[113] Microsoft. FASTER - Key Features. `https://microsoft.github.io/FASTER/`. Accessed: 2021-04-25.

[114] Microsoft. FasterKV basics - FASTER. `https://microsoft.github.io/FASTER/docs/fasterkv-basics/#checkpointing-and-recovery`. Accessed: 2020-12-07.

[115] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42, 2016.

[116] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94—-162, March 1992.

[117] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[118] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[119] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

[120] The nix-rust Project Developers. nix — crates.io: Rust package registry. `https://crates.io/crates/nix`. Accessed: 2020-04-15.

[121] 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, page 8–17, New York, NY, USA, 1988. Association for Computing Machinery.

[122] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[123] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017.

[124] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. *IACR Cryptology ePrint Archive*, 2016:916, 2016.

[125] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 3–33, 2018.

[126] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[127] Pouria Pirzadeh, Jun'ichi Tatemura, Oliver Po, and Hakan Hacigümüs. Performance evaluation of range queries in key value stores. *J. Grid Comput.*, 10(1):109–132, 2012.

[128] Serguei Popov. The Tangle. `https://www.iota.org/research/academic-papers`. Accessed: 2018-04.

[129] The GNU Project. Asynchronous I/O (the GNU C library). `https://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html`. Accessed: 2020-04-15.

[130] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[131] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making authenticated storage faster in Ethereum. In Ashvin Goel and Nisha Talagala, editors, *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX Association, 2018.

[132] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, pages 88–102, 2005.

[133] Michael K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5-9, 1994, Selected Papers*, pages 99–110, 1994.

[134] Ronald Rivest and Adi Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *Security protocols*, pages 69–87. Springer, 1997.

[135] Team Rocket. Snowflake to Avalanche: a novel metastable consensus protocol family for cryptocurrencies. `https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV`. Accessed: 2018-05.

[136] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. *CoRR*, abs/1906.08936, 2019.

[137] Rodrigo Rodrigues, Petr Kouznetsov, and Bobby Bhattacharjee. Large-scale byzantine fault tolerance: Safe but not always live. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, 2007.

[138] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[139] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5, 2014.

[140] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Optimizing flash-based key-value cache systems. In Nitin Agrawal and Sam H. Noh, editors, *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[141] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 207–220, 2000.

[142] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 169–184, 2009.

[143] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

[144] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Cryptography and Data Security, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, pages 507–527, 2015.

[145] Yonatan Sompolinsky and Aviv Zohar. PHANTOM: A scalable blockdag protocol. *IACR Cryptology ePrint Archive*, 2018:104, 2018.

[146] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 438–450, 2008.

[147] The Rust Team. Rust programming language. `https://www.rust-lang.org/`. Accessed: 2020-05-26.

[148] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[149] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. Karma: A secure economic framework for peer-to-peer resource sharing. In *Workshop on Economics of Peer-to-Peer Systems*, volume 35, 2003.

[150] Bitcoin Wiki. Difficulty - bitcoin wiki. `https://en.bitcoin.it/wiki/Difficulty`. Accessed: 2021-06-12.

[151] WonderNetwork. Global ping statistics: Ping times between WonderNetwork servers. `https://wondernetwork.com/pings`. Accessed: 2018-04.

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

[153] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association, 2015.

[154] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus in the lens of blockchain. *CoRR*, abs/1803.05069, 2018.

[155] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, 2019. ACM.

[156] Maofan Yin, Hongbo Zhang, Robbert van Renesse, and Emin Gün Sirer. CedrusDB: Persistent key-value store with memory-mapped lazy-trie, 2020.

[157] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336. ACM, 2018.

[158] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. S3: A scalable in-memory skip-list index for key-value store. *PVLDB*, 12(12):2183–2194, 2019.