BUILDING DATABASES WITH DISTRIBUTED TRUST

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

Florian Suri-Payer

August 2025

BUILDING DATABASES WITH DISTRIBUTED TRUST

Florian Suri-Payer, Ph.D.

Cornell University 2025

Modern distributed systems involve a diverse set of participants—ranging from cloud providers to jurisdictions, organizations, and individuals—who need to share data without necessarily trusting one another. These systems must ensure data availability and integrity, even when parties have disjoint, selfish, or adversarial interests. Byzantine Fault Tolerant (BFT) protocols provide strong guarantees in such settings and, for example, underpin much of today's blockchain infrastructure. However, existing BFT solutions often fall short, delivering poor performance and rigid, restrictive interfaces.

This dissertation proposes a new approach to efficient data sharing in environments with distributed trust—one that combines the robustness of BFT protocols with the performance and flexibility of traditional databases. We challenge the conventional BFT architecture, which centers on constructing a shared, tamper-proof totally ordered log and layering transactions on top. Instead, we advocate building a partially ordered BFT datastore directly. In particular, we argue that BFT systems, like traditional databases, should guarantee only *serializable* executions—those equivalent in effect to some total order—thereby avoiding the overhead of explicit total ordering.

We realize this approach through two systems: Basil and Pesto. Basil is a distributed BFT key-value store that integrates replication and transaction coordination into a single, low-latency architecture. It adopts a client-driven design, enabling parallel and independent transaction execution and improving robustness over traditional BFT protocols. To support richer application needs, Pesto extends Basil with a SQL-style query interface, allowing seamless integration with existing systems.

BIOGRAPHICAL SKETCH

Florian was born on March 10th, 1997, to Neeraj and Elisabeth—possibly on a dark and stormy night, though it might as well have been sunny (we'll never know, and you're not going to check). After bravely fending off oversized rodents in a Princeton daycare, he led his family first to Boston, and then onward to Gothenburg, Sweden. There, he was joined by his smaller counterpart, Fabian, whom he continues to raise in his likeness (though Fabian is notably three centimeters shorter).

Florian spent three formative years studying the adventures of the Gummi Bears and building snow castles, before relocating to Heidelberg, Germany to pursue the serious studies of elementary school. It was there that Florian's math skills peaked—since then, Florian has grown larger, but not necessarily wiser. Outside the classroom, Florian spent much of his time playing tennis and swimming.

After graduating elementary school summa cum laude, Florian once again led his family to a new home, this time in Hanover. There, he embraced the slightly less serious studies of high school while continuing to play tennis—and now also football (the kind where you actually kick the ball). Though he always loved the sciences, especially physics and chemistry, he chose to study computer science for university, which led him to Berlin. It was in Berlin that Florian met Clara, a beautiful girl who, perhaps in a moment of misjudgment, offered him an orange-flavored Maoam candy (as is well known, orange is the least popular flavor, easily outshined by cola and raspberry). They've been together ever since. Because Florian clearly cannot resist moving, he next set off for the gorgeous hills of Ithaca, New York to begin his PhD at Cornell. PhDs have a habit of being long, and Florian now seems to remember little of his life before. Equal parts difficult and joyful, these years have left a deep mark—and many memories. Now, as he graduates a *dottore*, Florian prepares to leave Ithaca behind and embark on his next grand journey: the quest to finally conquer basic calculus.

To Mom, Dad, Fabian, and Clara

Thank you for your constant support, love, and belief in me.

ACKNOWLEDGEMENTS

I am endlessly grateful to my advisors, Natacha Crooks and Lorenzo Alvisi. Both of them are exceptional people, and I could not have been luckier to experience their mentorship, kindness, and friendship throughout my PhD—and even before.

I first met both Lorenzo and Natacha over a Skype call (this was before Zoom took over the world, after all) back in 2017, as a wide-eyed second year undergraduate. Somehow, they agreed to host me for a brief six-week visit at Cornell. I didn't know it then, but that would be the first of many, many meetings to come. I remember being in complete awe of Cornell during that visit. Everyone I met was incredibly smart, yet more welcoming and kind than I had experienced throughout school before. What a strange and wonderful place Cornell is: world-class professors whose doors are always open, who treat you like an equal. I probably contributed very little during that brief visit, and yet both Lorenzo and Ken Birman wrote me letters for graduate school applications. It was an easy decision to come back to Cornell.

Lorenzo is perhaps the most Italian man I've ever met. He proudly wears custom Italy-flag Converse shoes, willingly drives an Alfa Romeo that breaks down annually, and is ready to fight you should you dare to break pasta. Few people manage to radiate enthusiasm while their world is on fire—yet Lorenzo does it effortlessly. Lorenzo recently became the department chair, a role which he *never* complains about—not even once a day. If you know Lorenzo you can still hear his iconic "Yahooooooo!" echoing in your head. Truly, few people have catchphrases that are so memorable. Lorenzo strikes a rare balance between being deeply critical and incredibly encouraging. His standards are high—which makes us a good match—but he always finds ways to be reassuring, motivating, and inspiring; even when Reviewer 2 is once again at his worst. I've benefited immensely from his rigor, both in technical discussion and writing (no comma or *e.g.*, is safe when Lorenzo makes his pass). Though, granted, it may take him 14 hours

to read through an introduction section. What has meant most to me, though, is our relationship beyond research. We've spent countless hours discussing football—clearly the European kind—and Lorenzo is always one to appreciate the low-quality memes that now proudly adorn the sacred walls of SysLab. Lorenzo and his amazing wife Irene treat us students as family. Every visit to their home is a cherished memory I'll carry with me.

As enthusiastic as Lorenzo is, so quippy is Natacha. Natacha is, quite literally, a French *and* British woman, which surely must be an oxymoron. A force for change wherever she goes (thankfully, these revolutions are peaceful), Natacha remains committed to building inclusive, thoughtful communities—yet somehow continues to mispronounce the word "schedule" (it's "skeh-jool", Natacha, not "shed-yool"). Natacha has been—and continues to be—a role model throughout my PhD and beyond. She is incisive, compassionate, and funny. She has a remarkable ability to grasp the big picture while attending to the tiniest detail, and she's taught me a great deal about explaining complex topics clearly. What stands out most, though, is her unwavering support for her (and others') students—not just academically, but as people. She puts an enormous effort into creating an environment where everyone feels welcome, not just those in the "inner circle" of a lab, conference, or community. Natacha is the person I go to most often for advice—technical, career, or just coping with disappointment. Her guidance has been invaluable to my growth and every success I've had. Sadly, none of this makes up for her appalling taste in football teams. While she is a proud supporter of Paris Saint-Germain—a club now synonymous with money and mediocrity—I hold out hope that one day she'll see the light and support a team that actually plays football. Say, Bayern Munich.

To Lorenzo and Natacha, from the bottom of my heart: thank you.

I also want to thank Ken Birman, who has been a champion for me—and for my partner Clara—even before I began my PhD. Ken has consistently gone out of his way to advocate for my career and personal growth, and I'm incredibly grateful for his unwavering support. He's been both a mentor and a friend throughout my time at Cornell, always enthusiastic about discussing ideas and offering guidance.

I'm also thankful to my committee members, Andrew Myers and Will Cong, for their invaluable feedback and support. Andrew is often the smartest person in the room, yet—like so many professors at Cornell—he is incredibly humble and approachable. I'm grateful for his sharp eye and technical insight, and I've been lucky to witness his unmatched puzzle-solving abilities firsthand. Will has also been a great mentor, always engaging with my work and offering a refreshing perspective from the world of finance and entrepreneurship. I particularly enjoyed helping Will teach a blockchain primer for his FinTech class, a rewarding opportunity to learn how to explain complex systems to a broader audience.

My thanks also go to Fred B. Schneider and Robbert van Renesse, who have each influenced my development over the years. Few people attend and contribute to our weekly Systems Lunch seminars as consistently and insightfully as Fred and Robbert. What seems routine for them is indispensable to us students. Fred, in particular, has an uncanny ability to cut through the noise and pinpoint the essence of a research problem—often to the dismay of the presenter. Though his questions can be intimidating, I've always found him to be kind and supportive. His sublte smile is iconic, and no compliment has meant more to me than when Fred once told me "Good job" after a talk. Robbert, meanwhile, is your friendly neighborhood systems guru. He still codes in a green-on-black terminal, and he’s always game for a deep dive into the latest systems idea. I vividly remember Robbert explaining over dinner how he had recently created a formally verified language for synchronization primitives, which required compressing pointers into 37 bits. When I asked if this took “two yea...”, he casually interjected, “Yes—about two months”. Robbert is a living legend, and no bit is safe when he’s scouring the block.

I also want to thank all my collaborators, who have been instrumental in shaping the research in this dissertation. In particular, I’m grateful to Neil Giridharan and Matthew Burke, my closest collaborators during the PhD. Their insights, creativity, and friendship have made this journey not just productive, but enjoyable. I’m lucky to have worked with them.

To my labmates, friends and co-conspirators—thank you for the many discussions, debates, and laughs we’ve shared. You’ve been a constant inspiration and support. In no particular order, I want to thank: Soumya Basu, Burcu Canakci, Shobhita Gupta, Matthew Burke, Yunhao Zhang, Ted Yin, Cong Ding, Sowmya Dharanipragada, Youer Pu, George Karagiannis, Niko Grupen, Gloire Rubambiza, Kevin Negy, Neil Giridharan, Audrey Cheng, David Chu, Shir Cohen, Yu-ju Huang, Silei Ren, Yulu Yao, Austin Li, Muhammad Ahmed, Shubham Chaudhary, Ali Farahbakhsh, and Suraaj Kanniwadi.

Soumya has been both a great friend and mentor. Choosing him as my first-year mentor was one of my better decisions. I’ve learned a lot from Soumya—sometimes from his advice, and sometimes by doing the exact opposite of what he would do. He’s like that clumsy older cousin you can’t help but love: too lazy to tie his shoe laces or even take the stairs, yet always there when you need him, and incredibly tenacious once he’s made up his mind. Even though he’s *not exactly* a football prodigy, you always

want him on your team.

Burcu is one of the most fun people I've ever met. We first bonded during a summer internship at Microsoft Research in Cambridge. She's an (allegedly) retired professional swimmer and an elite canal punter—never once has she ended up in the water. One time, she tried to have some cows kill me (true story). Back at Cornell, I fondly remember playing card games with her, Soumya, and Shobhita, where she'd go to great lengths to make sure anyone but me won. Burcu's humor is infectious, and I miss having her around.

Matt, Yunhao, Cong, Youer, Sowmya, and I were all part of Lorenzo's group during my early years at Cornell. All five of them are unique and talented individuals, and I am grateful for the time we spent together. We shared many a celebratory ice cream at the world-famous Cornell Dairy Bar after successful paper submissions. I fondly remember how Cong, Matt, Yunhao, and I once drove seven hours to Canada to attend SOSP—spotting Lorenzo on his motorcylce along the way. Sowmya joined the group the same time I did and is perhaps the most negative person I know—yet never fails to make me laugh. I'll always remember the chaos of Lorenzo's class projects and the spirited arguments with Cong and Youer (who were our TAs) over our homeworks.

I joined the PhD program alongside George, Niko, Gloire, and Kevin, and I'm glad to have shared classes, lab space, and many memories with them. George and Niko were the first friends I made at Cornell, and I vividly remember jokes, lunches, and pickup games we shared. Gloire, a man of many hairstyles, is one of the most inspiring people I know: he's overcome extraordinary adversity and remained unshakably kind. Kevin, meanwhile, is one of the most pleasant and genuine people you'll meet. I especially appreciate his honesty when it comes to mental health—something that can be difficult to talk about, especially during the early years of a PhD. He's also far better at basketball than I am, and to this day refuses to retire his trusty, half-decrepit ball.

To Neil, Audrey, David, and the rest of SkyLab at Berkeley: thank you for welcoming me during my visit. Neil and I went on to collaborate extensively and played an unholy number of table tennis matches together. He once beat me 23 times in a row. Naturally, we kept playing until I won once (it took a few hours). Neil is a great friend, and I am grateful for the many discussions we have had both about research, and all things sports. Audrey and David are immensely talented, though Audrey works much too hard. You wouldn't guess it by talking to her, but Audrey is a world class Pokémon Unite athlete (though the accuracy of this memory is a bit hazy). David, meanwhile, is a professional foodie. Ask him where to eat in Berkeley.

To my current labmates—Yu-Ju, Silei, Yulun, Shir, Austin, Muhammad, Shubham, Ali, and Suraaj—thank you for being such an incredible group. Yu-Ju is the most chill researcher I know: never in a rush, always composed. Silei, in contrast, is a whirlwind of energy and always up for a spontaneous plan. Nobody really knows what Yulun is up to—he appears out of nowhere, sometimes with a beach chair in tow, and vanishes just as quickly. Shir is the only person I know that is more comfortable coding in bash scripts than any *normal* programming language. I still don't fully understand how she does it, but somehow, it always works. Austin and Muhammad are our newest additions, and I'm excited to see what they bring to the lab. They've got much to learn when it comes to contributing memes, but I'm sure they'll catch up quickly. Shubham is the lab's unofficial uncle—always ready with a funny story or obscure cultural reference. He's also an extremely gifted engineer who's saved me more than once from hair-pulling bugs. Ali is the most stubborn person I know, and Suraaj the most unhinged—in the best possible way. Together, the combination is unstoppable. Suraaj, too, is a fantastic engineer and has helped me debug more issues than I care to admit. Ali, meanwhile, lives in the clouds, always chasing the most obscure and esoteric research ideas. Their meme judgment is harsh (not everyone understands art!), but their friendship and energy made every day brighter. I'll miss them both dearly.

I'm also grateful to my friends outside academia, who kept me grounded, laughing, and sane. Thank you to Jannik Foerster, Julius Coburger, Vincent Borko, Daniel Holland, Sam Heinrich, April Tan, Kyrin Pollack, Alex Miaule, Alex Hayre-Pérez, Rohan Bansal, Olu Aliweri, and Shiv Salwan.

Jannik is my oldest friend—we have known each other since we were 12. He's one of the kindest and most caring people I know, and I'm deeply grateful for his friendship. Ironically, Jannik started out quite the opposite: in high school he was a fierce football sweeper. Where other kids might kick under the table, Jannik would punch above. He's also the most competitive person I know, and we have spent many hours competing in every sport we come across (I usually win). I'll never forget our table tennis duels, especially how Jannik always managed to hit the ball into the net.

Julius and Vincent were my closest friends during my undergraduate at TU Berlin, and we've remained close ever since. I am incredibly grateful for their friendship, and I fondly remember the many joyous experiences we shared—and continue to share. Julius is incredibly smart, but hilariously clumsy, making him an easy target for jokes. I will never forget him eating raw cereal straight from the box in the Berlin subway, or falling asleep across *four* chairs during a lecture. Vincent, on the other hand, is the laziest brilliant person I know; I'm not sure whether he ever completed a single homework assignment. To the many burgers we shared, the countless hours of video games, and the endless laughter: *dopa down*.

Being part of the Cornell Club Tennis team was truly one of the most memorable parts of my time here. We shared many great moments on and off the court, and I deeply appreciate the camaraderie and friendships we've built together. In particular, I want to thank Rohan, Daniel, Sam, *little* Alex, and *big* Alex—both Alexes are actually the same

height, but you'll know immediately who's who. Rohan, Daniel, and Sam were my first friends on the team; their friendship made settling into Cornell—and the U.S.—much easier. Traveling together to tournaments (and winning!) was some of the most fun I've had. I especially remember our trip to Delaware—where Rohan was most definitely *not* hungover during a match. Since then, I've watched him grow from a goofy freshman into an even goofier adult. I fondly remember movie nights with Sam, Daniel, April, and Kyrin, and especially Daniel's consistent dominance in trivia games. Over the past four years I've also had the pleasure of getting to know *little* and *big* Alex—both incredibly talented and fun to be around. I'll miss all of our tennis sessions, though I'll forever despise Sam's banana shots.

Getting to know Olu and Shiv was one of the unexpected highlights of my time at Cornell. Both of them are simply incredible people, and two of the most kind and supportive friends I've been fortunate to meet. Although neither has a technical background, they've consistently been the most curious and engaged in my work—something I truly appreciate. They've taught me a great deal about life and culture from different perspectives, and I am deeply grateful for their friendship. I fondly remember the many tennis matches we attended, and all the free merch we picked up along the way.

Finally, I want to thank and acknowledge my family and my partner, Clara. My parents, Neeraj and Elisabeth, instilled in me a sense of curiosity and a love for learning that has been invaluable throughout my life. They have supported me in every endeavor, and I am impossibly grateful for their unwavering support and encouragement. My brother Fabian has also been a constant source of love and friendship. He pushes me every day to live up to the person he believes I can be—and that he is proud of. Clara is the most amazing person I know. She has been there for me through the ups and downs of my PhD journey, and I am endlessly thankful for her patience, encouragement, and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Modern distributed systems consist of a variety of participants—from cloud providers to jurisdictions, organizations, and individuals—that all want to share data. Such systems must maintain the availability and integrity of the data, even when different parties have disjoint or selfish interests, and lack trust in any one party. Byzantine Fault Tolerant (BFT) protocols offer datastores such a promise; for instance, they lie at the heart of today's blockchain infrastructure. Existing solutions however, fall short, providing both lackluster performance and restrictive application interfaces. In this dissertation, we advocate for a new approach to efficient data sharing with distributed trust that combines the robustness of BFT systems with the efficiency and flexibility of traditional database systems.

1.1 Distributed Databases with Limited Trust

Modern web applications must be highly *available*. Users expect continuous, uninterrupted service—often measured in multiple "nines" of reliability—and even brief downtimes can be costly, risking user frustration and eroding goodwill. To ensure reliable uptime, application providers rely on *replication*: deploying redundant copies (*replicas*) of the underlying data store ensures that applications remain available even as some replicas fail. These replicas may be located within the same datacenter—also aiding with load balancing— or distributed across geographically distinct regions to mitigate regional outages and to improve access latency.

However, operating multiple data copies introduces concerns around *consistency*. High availability is of little value if the underlying data is not *correct*. As such, applications often seek strong consistency guarantees: users should observe a coherent and up-to-date view of the system regardless of which replica they interact with, and the system must tolerate failures without risking data loss or corruption.

In addition to being available and correct, web applications must be *fast*. For instance, Amazon once famously reported a 1% drop in sales for every additional 100 ms of latency experienced by users [70]. Google likewise observed a 20% drop in user traffic when page load times increased by just 500 ms [70, 117]. To meet these performance demands, applications strive to (*i*) minimize coordination during replication, (*ii*) execute operations in parallel where possible, and (*iii*) scale out horizontally with ease.

To ensure *correct* parallel execution, developers rely on transactions. Transactions guarantee atomicity, sparing developers from the complexity of reasoning about interleaved operations. For scalable execution, data stores are further partitioned across (replicated) *shards*, allowing independent, commutative operations to proceed concurrently without interference.

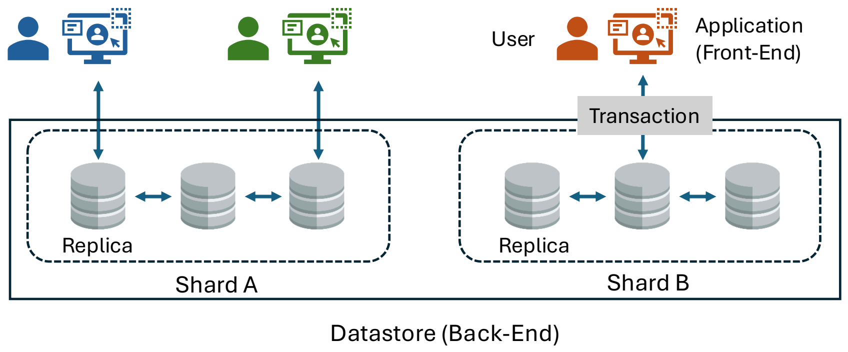

Figure 1.1: A simple distributed database architecture. Multiple users issue transactions concurrently, which are routed to the appropriate shard based on the requested data. Within each shard, users interact with their nearest replica, while state changes are consistently replicated across all replicas within that shard.

This *distributed database* architecture (Fig. 1.1) is well suited for settings with a single administrative domain, where a single organization, jurisdiction, or individual operates the data store. It provides high availability, strong data integrity, and good performance. A prominent example is Spanner [40], Google's flagship distributed data storage system, which underpins many of its applications and serves billions of users.

Modern applications, however, increasingly require the ability to *share* data across organizational boundaries. The architecture described above does not extend gracefully to multi-administrative-domain environments, where mutually distrustful parties seek to jointly operate a service.

Consider, for instance, a consortium of banks (illustrated in Figure 1.2) aiming to build a decentralized payment infrastructure, thereby bypassing traditional centralized clearinghouse networks [13]. Although no single bank fully trusts the others, they must still coordinate and share resources to support the joint system.

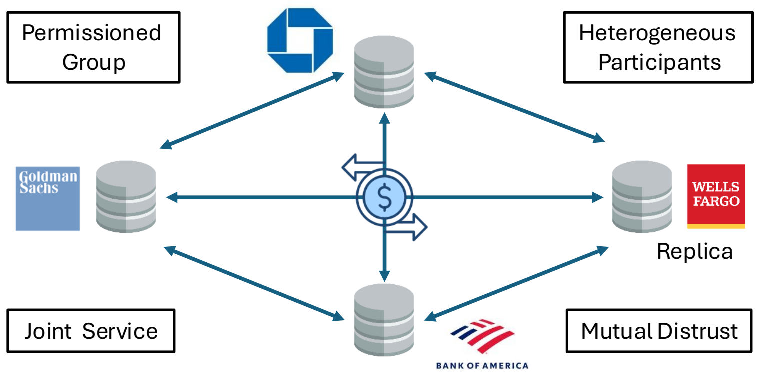

Figure 1.2: A simple banking consortium. Four mutually distrustful banks aim to provide a joint transaction clearing service.

Unfortunately, existing distributed database solutions are typically designed to tolerate only *crash* failures, where nodes simply stop responding. They do not account for more severe adversarial behaviors, such as participants that deliberately violate protocol semantics—by lying, equivocating, or attempting to corrupt the system's state.

Byzantine Fault Tolerance (BFT) [111] offers a solution: BFT systems aim to ensure consistency even in the presence of a subset of misbehaving (or *Byzantine*) participants. Originally developed to improve the resilience of mission-critical infrastructure—such as spacecraft control systems or national power grids—BFT techniques are now widely adopted in modern blockchain platforms [13, 60, 62, 64, 65, 119]. These systems enable new forms of decentralized coordination and offer promising opportunities across diverse domains, including healthcare [118], financial services [13, 58, 87], and supply chain management [47, 186].

1.1.1 BFT Deployment Challenges

At the heart of most existing BFT solutions lie replicated state machines [30, 35, 80, 101, 192]. They offer the abstraction of a shared, tamper-proof, totally ordered log: all participants agree on the same sequence of operations—even if a subset of participants is Byzantine. This abstraction greatly simplifies the task of building consistent application state on top. For example, in the context of our consortium of banks, account balances can be deterministically computed from the log, allowing the system to behave as though it were a trusted, centralized database (Fig. 1.3).

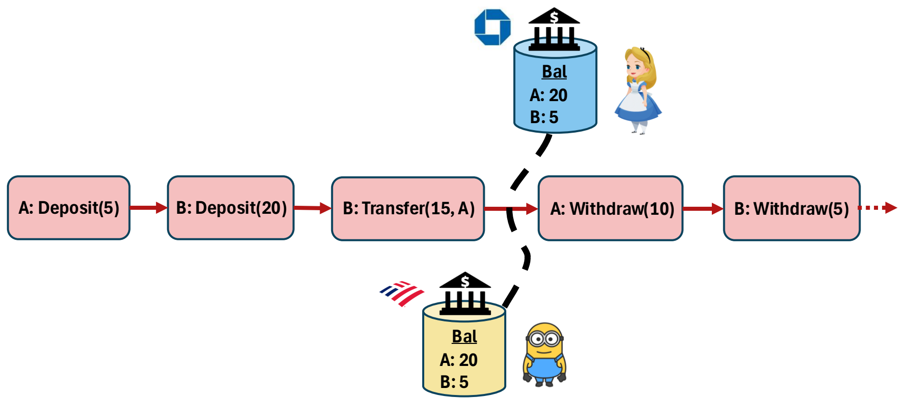

Figure 1.3: The shared totally ordered log used by the banking consortium. Alice and Bob are customers at different banks but observe the same, consistent view of the world.

While this abstraction is powerful and desirable for building applications, implementing it in a way that is efficient, robust, and developer-friendly remains a significant challenge.

Performance Shortcomings While replicated state machines offer strong consistency in the presence of Byzantine faults, they come with substantial performance costs. Most notably, BFT protocols suffer from high latency. Establishing a global total order requires multiple rounds of coordination, each involving resource-intensive cryptographic authentication (*e.g.*, signature generation and verification). The two most widely used BFT state machine replication protocols—PBFT [30] and HotStuff [192]—require 5 and 9 message delays, respectively, before responding to a client request. These delays compound significantly in *interactive* transactional workloads, where each operation may depend on the result of the previous one.

Total ordering also imposes a throughput bottleneck. Since all operations must be serialized, much of the inherent parallelism in a transaction workload is lost. This is inefficient, as many transactions are commutative and could safely execute in parallel. Consider, for example, Alice and Bob purchasing a Ferrari and an ice cream, respectively—these transactions touch disjoint objects and yield the same final state regardless of their execution order.

While software transactional memory (STM) techniques [49, 68, 162] can help recover some parallelism, they introduce additional complexity and still require establishing an initial total order.

To mitigate this, some BFT systems adopt sharding [7, 98, 145, 146, 194], allowing transactions that access disjoint shards to proceed concurrently. However, this too is a limited remedy. Within each shard, operations must still be totally ordered. Cross-shard transactions require coordination via Two-Phase Commit (2PC), which itself involves additional ordered steps—introducing further latency. Moreover, sharding is not always practical. Arbitrary sharding can lead to increasingly thin partitions, where transactions span many shards. In such cases, cross-shard coordination overheads grow linearly with the number of involved shards—incurring substantial cryptographic and communication costs in a BFT setting.

Finally, BFT replicated state machines exhibit poor load balancing, limiting scalability under high demand. Because all replicas must redundantly execute every request, resource utilization is poor. This is particularly problematic in BFT systems, which already require higher levels of redundancy to tolerate faults.

Brittle Robustness BFT protocols typically rely on a dedicated *leader* or *sequencer* to establish the total order. This centralization creates both a performance bottleneck, and a point of fragility. All requests must pass through the leader, which can throttle the system's throughput. While recent protocols decouple data dissemination and scale it across all replicas [73, 165, 166], the leader remains essential for coordinating agreement and ensuring progress.

This design raises both fairness and robustness concerns. The leader has disproportionate control over transaction ordering and may censor or reorder transactions to its advantage—*e.g.*, by executing a sandwich attack in finance applications [44]. Furthermore, the leader is a single point of failure. Its crash or slowdown can halt the system until a new leader is elected. Recent research [73] shows that even temporary interruptions can cause longer-term performance degradation, despite recovery.

Lack of Programmability Modern applications favor interactive transactions, which simplify development by allowing developers to express complex workflows incrementally and intuitively, while still ensuring strong correctness guarantees [40, 149]. The designers of Google Spanner, for instance, argue that "it is better for application programmers to deal with performance problems due to overuse of transactions [...] than always coding around the lack of transactions" [40].

Unfortunately, the high latency inherent in BFT protocols renders interactive transactions largely impractical. To offer reasonable performance, BFT systems often resort to simplified transaction models: assuming fixed structure [146], requiring advance knowledge of read/write sets [17, 145, 146], or restricting execution to a *one-shot* model—*i.e.* a single request with no interaction [146]. Most blockchain systems, for instance, rely on stored procedures (commonly called *smart contracts* [198]), which execute entirely server-side without further client input [16, 60, 62, 64, 65].

This limits programmability. Applications must be written in a domain-specific language (DSL), complicating both development and maintenance. Updating logic requires carefully coordinated deployments to ensure consistency across replicas.

Ironically, this server-centric model can also hinder scalability. Although interactive transactions incur higher latency, they allow most application logic to be executed *client-side*—*e.g.*, in stateless frontends that can scale independently of the replica set and without requiring (expensive) consensus coordination. In contrast, stored procedures concentrate all execution at the replicas, creating scalability bottlenecks under load.

1.1.2 Contributions

In this dissertation, we propose a more principled, performant, and expressive approach to realizing the abstraction of a Byzantine fault tolerant (BFT) totally ordered log. We draw on the lessons of classical databases, which use interactive transactions to efficiently implement the *abstraction* of a sequential, general-purpose BFT log. In these systems, transactions need not execute in strict sequence; rather they must produce executions that are *equivalent* to some serial schedule—a property known as *serializability* [22, 147]. This ensures correctness while allowing for high concurrency: ordering is required only for conflicting operations.

We argue that BFT systems can—and should—adopt a similar philosophy. Rather than enforcing a total order and layering database-like transactions on top, we propose to *decouple* the abstraction of a totally ordered log from its implementation. Our approach directly realizes the log abstraction on top of a partially-ordered distributed database.

This shift enables both better performance and improved robustness. For example, replicas no longer need to agree on a total order upfront, and can therefore avoid using a leader. However, giving up total order introduces new challenges. How can the system reliably (and correctly) resolve conflicts when operations do not commute? Without a globally agreed-upon order, correct replicas may process transactions in different orders—and diverge. Worse, misbehaving participants may exploit this ambiguity to attack the system's integrity. Users cannot trust any single replica to produce a correct result, yet even correct replicas may (legitimately) disagree. How then can clients receive trustworthy results and make reliable progress?

This dissertation addresses these questions through the design and implementation of two systems, Basil and Pesto, which both *scale the abstraction of a totally ordered* *log*—the former as a BFT key-value store, and the latter as a full BFT SQL database. Our work contributes the following:

1. Formalizing correctness for transactional BFT applications We begin by addressing a foundational question: what does *correctness* mean in a Byzantine setting?

At a high level, our system must preserve the illusion of a sequential execution. For instance, two concurrent transactions accessing the same data should appear to execute in isolation and should never observe inconsistent views. Additionally, the system must be robust to Byzantine behavior: malicious participants should neither violate correctness, nor prevent progress.

However, Byzantine participants can also submit transactions. What guarantees should hold for these? And how should their effects be perceived by correct participants?

We propose two complementary correctness properties:

- **Byzantine Isolation**, which ensures safety. Correct clients must observe a data state that could have resulted from the actions of correct clients alone. Byzantine clients may violate isolation for their own transactions, but cannot cause correct participants to observe inconsistent state.
- **Byzantine Independence**, which ensures progress. It bounds the influence that Byzantine actors can exert on the success or failure of operations. Byzantine clients can harm their own transactions but cannot impede those of correct clients.

2. Basil, a scalable BFT key-value store To scale the abstraction of a totally ordered log, we present *Basil*, a distributed BFT key-value store that supports serializable, interactive transactions with high performance and robustness.

Basil draws inspiration from crash fault tolerant distributed databases [138, 195], and avoids imposing a total order. Instead, it integrates replication and transaction coordination (*e.g.*, concurrency control and Two-Phase Commit) into a unified, low latency protocol.

To uphold Byzantine Isolation and Independence, Basil embraces the principle of *independent operability*: safety and liveness are enforced on a per-client, per-transaction basis. Clients are responsible for driving the execution and commit of their own transactions, and proceed independently and in parallel. Basil uses optimistic concurrency control to detect and resolve conflicts. In the absence of contention and faults, clients can commit transactions in just a single round-trip.

Our evaluation shows that Basil outperforms traditional order-based BFT systems by 4-5x in throughput, while achieving performance within 4x of TAPIR [195], a leading crash fault tolerant (CFT) system. Crucially, Basil remains robust under failure: faulty clients affect only their own or conflicting transactions—and correct clients can safely resolve stalled conflicting transactions themselves. With 30% Byzantine clients, throughput degrades by less than 25%.

3. Pesto, a plug-and-play BFT SQL database While Basil delivers robustness and performance, it lacks support for expressive query interfaces. To bridge this gap, we present *Pesto*, a general-purpose BFT SQL database that builds on Basil and adds support for rich queries.

Supporting SQL-style queries—without total order—in a Byzantine setting introduces new challenges. First, ensuring serializability for arbitrary queries requires trusting the correctness of computation performed by potentially faulty replicas. Yet, due to asynchronous execution and partial orderings, even correct replicas may produce different—but still valid—results. This makes it difficult to determine the integrity of query results. Second, optimistic concurrency control traditionally performs poorly for range queries, which must—at least logically—lock all keys the query accesses, reducing concurrency.

Pesto addresses both challenges with a key insight: query execution requires consistency and conflict-detection only over the *predicate* relevant to the query—not the entire database. Pesto leverages this insight to implement a novel, client-driven snapshot synchronization mechanism which ensures that replicas agree on the relevant subset of data for a given query. To minimize transaction conflicts, Pesto employs a predicate-aware optimistic concurrency control scheme that only aborts concurrent transactions that violate query semantics.

Our evaluation shows that Pesto significantly outperforms traditional BFT systems that layer SQL functionality on top of total ordering (*e.g.*, 2.3x higher throughput and up to 3.9x lower latency on TPC-C). It also performs competitively even with PostgreSQL [78], a widely used unreplicated production-grade SQL database—achieving equivalent throughput on TPC-C. Pesto's client-driven design minimizes the impact of replica failures and ensures robust performance.

1.2 Dissertation Overview

This dissertation is organized as follows. Chapter 2 provides foundational background on transactional databases and Byzantine Fault Tolerance (BFT). It introduces key concepts in transaction processing, surveys the architecture of modern distributed databases, and reviews representative BFT consensus protocols. Chapter 3 formalizes correctness criteria for transactional applications operating in Byzantine environments and presents *Basil*—a distributed BFT key-value store that achieves high performance and robustness by eschewing total ordering in favor of a parallel, client-driven architecture. Chapter 4 extends this approach to support richer query semantics. It presents *Pesto*, a distributed BFT SQL database designed to integrate seamlessly with existing applications while maintaining strong robustness and performance. Finally, Chapter 5 concludes the dissertation and highlights additional related work not covered in the main chapters.

CHAPTER 2

BACKGROUND

The previous chapter scoped the goal of this dissertation: to build a scalable Byzantine Fault Tolerant (BFT) database that provides high performance, strong robustness, and an expressive application interface. This chapter provides the necessary background on transactional systems, BFT, and distributed databases to understand the challenges and solutions presented in this dissertation.

2.1 Transactions

Databases provide a way to store, retrieve, and manipulate data in a structured manner. They allow users to access their data, perform queries, and execute *transactions.*

A transaction is a sequence of operations that appears to take effect *atomically*, or "instantaneously" [147]: transactions either complete entirely or have no effect at all, leaving the database in a consistent state. Transactions simplify application development by providing developers with strong consistency guarantees, even when operations execute concurrently. This makes them essential for applications like banking, e-commerce, and any system where data integrity is critical.

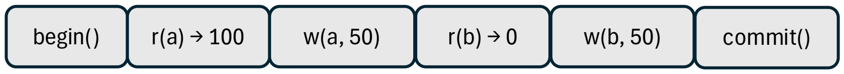

Figure 2.1: A transaction T defines an atomic sequence of operations. Here r denotes a read operation and w denotes a write operation.

Figure 2.1 illustrates a simple transaction that transfers money between two accounts. The payment transaction consists of two actions: deducting an amount from one account and adding the same amount to another account. If either operation fails, the entire transaction is rolled back, ensuring that no money is lost or created.

Traditionally, transactions are said to uphold the four **ACID** properties: **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- **Atomicity**: A transaction is an atomic unit of work, meaning that either all operations in the transaction are executed, or none are. If a transaction fails, the system is left in a state as if the transaction never occurred.
- **Consistency**: A transaction brings the database from one consistent state to another. It ensures that the database remains in a valid state before and after the transaction. Consistency is dependent on the invariants of the running application. It may be enforced either explicitly, using database features (*e.g.*, unique or foreign key constraints), or implicitly, via application-level checks (*e.g.*, the application logic ensures that the sum of balances remains constant).
- **Isolation**: Isolation defines a contract on how concurrent transactions interact. It determines when the effects of executing transactions become visible to each other, and what *anomalies* may arise. Weaker isolation constraints permit higher levels of concurrency—at the risk of violating integrity guarantees—while the strongest isolation constraints ensure that applications execute free of anomalies—at the cost of lower performance.
- **Durability**: Once a transaction is committed, its effects are permanent and survive system failures. The database guarantees that the changes made by a committed transaction will not be lost.

2.1.1 Serializability, the Gold Standard for Isolation

To ensure high performance, transactional databases permit concurrent execution of transactions. However, this concurrency can introduce anomalies if transactions interleave in undesirable ways. Improper interleavings may cause unexpected behaviors such as dirty reads, non-repeatable reads, and phantom reads, all of which can violate the consistency of the database state [18]. To prevent such anomalies, databases enforce *isolation*, which governs how transactions interact and what inconsistencies may occur. Modern database systems support multiple isolation levels, each offering a different balance between performance and consistency guarantees.

This dissertation focuses specifically on transactional systems that implement *serializability*, the most stringent standard isolation level [11, 18].[1] At a high level, serializability ensures that transactions produce results equivalent to some serial execution order. This allows developers to reason about applications as if transactions ran in complete isolation—avoiding anomalies and maintaining strong consistency of the database state—, while still benefiting from the performance gains of concurrency.

Formally, we define serializability as follows:

Let T_1, T_2, ..., T_n be a set of transactions. Each transaction T_i consists of a sequence of read and write operations, ending with either a commit or an abort. A transaction schedule is a (possibly interleaved) sequence containing all operations from T_1, T_2, ..., T_n. A *serial schedule* executes each transaction's operations contiguously, without interleaving—*i.e.*, all operations of T_1 occur before any of T_2, then all of T_2 before T_3, and so on. In such schedules, transaction executions are strictly non-overlapping.

[1]Strictly speaking, the strongest isolation level is *strict serializability* [4, 147], which enforces external consistency (*i.e.*, it preserves real-time ordering and causality). However, plain serializability is typically the strongest isolation level implemented in production database systems [37, 78, 143, 144].

Put differently, a schedule is serializable if it can be transformed into a serial schedule by reordering its operations in a way that preserves the outcome of their execution.[2]

Definition (Serializable) *A transaction schedule is serializable if all operation outcomes are equivalent to those of some serial schedule, i.e., as if the transactions had executed one after another without interleaving.*

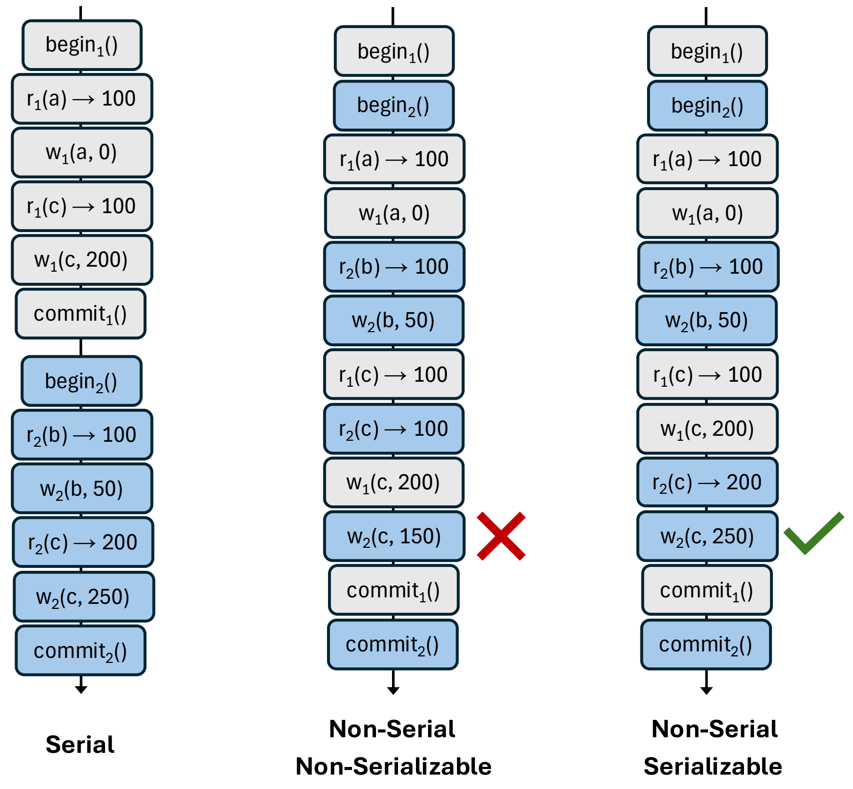

Figure 2.2: Serializability example. Two transactions, T_1 and T_2, execute concurrently. Their execution is serializable only if the outcome is equivalent to some serial order (e.g., T_1 before T_2), ensuring the database remains in a consistent state.

Example: Bank Transfer We illustrate serializability with a simple example involving two transactions that transfer money between accounts. Consider a database with three accounts—owned respectively by users A, B, and C—each with an initial balance of \$100. Both users A and B owe user C money, and both end up executing their respective

[2]This characterization of serializability follows conflict serializability [22], a widely used and practically enforceable definition. Other, more general definitions—such as view serializability [189]—permit more complex interleavings but are harder to verify and thus less commonly implemented in practice.

payment transactions concurrently: A's transaction (T_1) transfers \$100 from A's account to C's account, and B's transaction (T_2) transfers \$50 from B's account to C's account. Each transaction consists of four operations: two reads to determine the current account balances (one to each account), and two writes to reflect the new balances.

Figure 2.2 illustrates three execution schedules of these two transactions. The first schedule is serial, where T_1 executes first, followed by T_2. In this case, the final balance of A's account is \$0, for B it is \$50, and for C it is \$250. This is consistent with the initial balances of all accounts and the intended transfer amounts.

The second schedule is non-serial and non-serializable. Both T_1 and T_2 read the initial balance of C's account (\$100), and then each writes their own updated value—\$200 (= \$100 + \$100) for T_1, and \$150 (= \$100 + \$50) for T_2—resulting in an inconsistent final state. In this example, the final balance of A's account is \$0, but the balance of C's account is only \$150: the \$100 transferred by A has effectively been lost!

The third schedule is non-serial, but it is serializable. Here, T_2's read of C's account reflects the result of T_1's prior write, resulting in a correct final state.

2.1.2 Concurrency Control

Serializability defines a strong isolation contract for transactions, ensuring that they appear to execute in a serial order. **Concurrency control** is the mechanism that implements this contract. It regulates the acceptable interleavings of concurrent transactions and ensures that transactions commit if and only if their execution is serializable.

We broadly classify two approaches to concurrency control (CC): pessimistic and optimistic CC.

Pessimistic Concurrency Control Pessimistic Concurrency Control (PCC) assumes that conflicts between transactions are likely and proactively prevents them by locking data items before access. This approach ensures that only one transaction can access a data item at a time, thereby preventing undesirable interleavings. To reduce unnecessary contention, PCC protocols typically distinguish between shared and exclusive locks: multiple transactions can concurrently acquire shared locks to read the same data item, while exclusive locks ensure that only one transaction can write to it at a given time.

The most well-known PCC protocol is *Two-Phase Locking* (2PL), in which transactions proceed through two distinct phases: a *growing phase*, during which they acquire locks, and a *shrinking phase*, during which they release them. Notably, 2PL allows transactions to release locks before they commit, which can improve concurrency but may lead to cascading aborts if other transactions observe the effects of a transaction that later aborts. *Strict Two-Phase Locking* (S2PL) is a widely used variant that requires transactions to hold all exclusive locks until they either commit or abort. This guarantees recoverability and prevents cascading aborts but comes at the cost of reduced concurrency.

While lock-based approaches enforce serializability, they can introduce deadlocks when concurrent transactions compete for locks in conflicting orders. To resolve deadlocks, PCC protocols typically implement deadlock detection and resolution mechanisms, such as wait-for graphs, canonical lock ordering, or timestamp based techniques (*e.g.*, *wound-wait* and *wait-die*) [158].

Figure 2.3 illustrates a simple example of two transactions executing under 2PL. The reads of transactions T_1 and T_2 may proceed concurrently by acquiring shared locks. However, when T_1 attempts to write, it must obtain an exclusive lock and thus must wait until the lock becomes available. To avoid deadlock, T_2 is forced to yield its lock and abort, since T_1 has already registered its intent to acquire the exclusive lock.

PCC is often overly conservative, guarding against potential conflicts that may never materialize. As a result, transactions can unnecessarily block while waiting for locks to be released, leading to reduced concurrency and degraded performance.

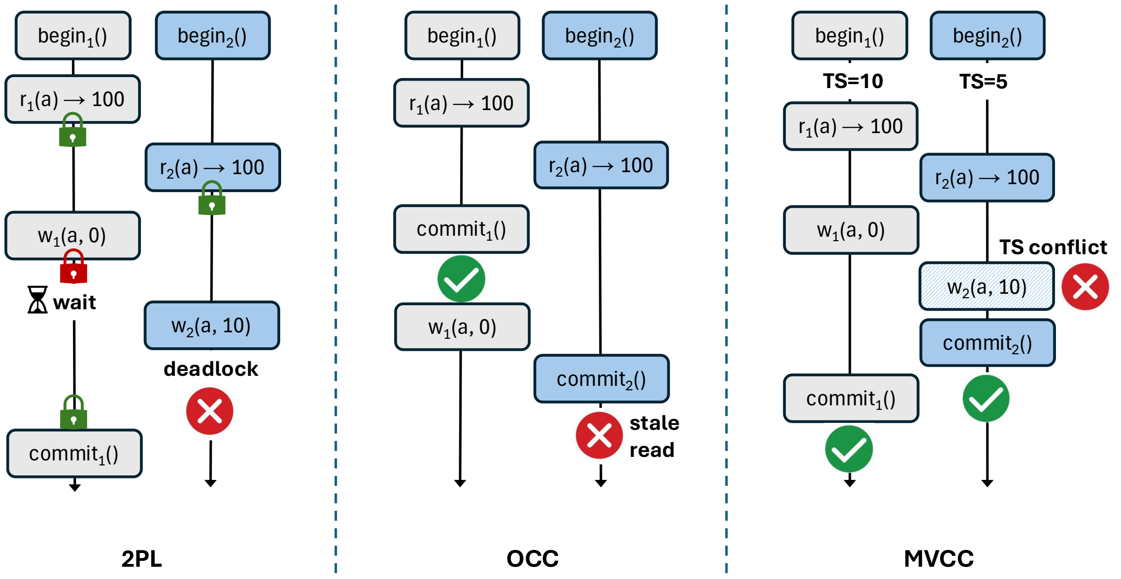

Figure 2.3: Illustration of conflict detection and resolution accross different concurrency control protocols.

Optimistic Concurrency Control Optimistic Concurrency Control (OCC), in contrast, assumes that conflicts between transactions are rare. OCC protocols allow transactions to execute speculatively without acquiring locks, and detect conflicts only at the end of execution through an additional *validation* phase.[3] This approach enables higher concurrency under low contention but can result in wasted work when conflicts do occur, as conflicting transactions must abort and retry.

In its simplest form, OCC buffers all writes in a transaction until it is ready to commit. At that point, it checks whether any of the data items it read have been modified

[3]In practice, validation is often piggybacked onto the final write or used to transmit previously buffered writes.

by other transactions. If so, the transaction aborts and retries; otherwise, it commits its writes. Figure 2.3 illustrates a simple example involving two transactions. The reads of T_1 and T_2 proceed without blocking, while their writes remain buffered. T_1 commits successfully—making its writes visible—after confirming that no concurrent writes have invalidated its reads. As a result, T_2's validation fails, as its earlier read is now stale.

More sophisticated OCC protocols incorporate timestamps, versioning, or other mechanisms to improve performance and reduce conflict likelihood. A widely used technique is *Multi-Version Concurrency Control* (MVCC) [20], which maintains multiple versions of each data item. This allows readers and writers to operate concurrently without blocking: readers access the most recent version as of the transaction's start time—effectively working on a consistent snapshot of the database—while writers generate new versions associated with a timestamp or version number.

Traditionally, MVCC enforces serializability during execution, though some protocols defer it to a validation phase [48, 171, 183]. In either case, writers must ensure that their updates do not conflict with the snapshots observed by concurrent readers. This typically involves checking the writer's timestamp or version number against the read versions of data items accessed by other transactions. If a conflict is detected, the writing transaction is aborted and retried. In practice, this is typically done by checking the timestamp (or version number) of a writing transaction against the snapshots observed (*read-versions*—timestamps or version numbers) by concurrent readers. If a conflict is detected, the transaction is aborted and retried.

MVCC provides high concurrency and avoids many of the drawbacks of locking-based approaches, such as deadlocks. However, it can lead to increased storage overhead, since multiple versions of each data item must be maintained. To address this, systems rely on garbage collection to reclaim storage from obsolete versions.

Figure 2.3 illustrates the running example under MVCC, using a timestamp-based conflict resolution strategy: Multiversioned timestamp ordering (MVTSO) [20]. In MVTSO, each transaction is assigned a timestamp at start time and reads the latest visible version smaller than its timestamp. A writer must abort if its write violates a reader's snapshot—*i.e.*, if the write falls between the version observed by the reader and the reader's timestamp. The figure illustrates transaction outcomes both with and without an optional write by transaction T_2.

2.1.3 Transaction Models

Transactions can be implemented in various ways. The choice of *transaction model* affects how transactions are executed, how isolation is maintained, and how concurrency control is enforced. This section briefly summarizes three models that are referenced throughout the dissertation.

Interactive Transactions This dissertation focuses primarily on *interactive* (sometimes called "general") transactions—*i.e.*, transactions that are interleaved with the client-side application logic and allow ongoing interaction between the client and the database system during execution. This is the most general and widely used transaction model. Developers favor interactive transactions [149] because they simplify programming: only database requests must be expressed in the database's domain specific language (DSL), while control logic stays in application code. This model enables applications to issue requests dynamically, based on intermediate results or external inputs (*e.g.*, user input, or real-time feedback). Figure 2.4 illustrates a simple interactive transaction.

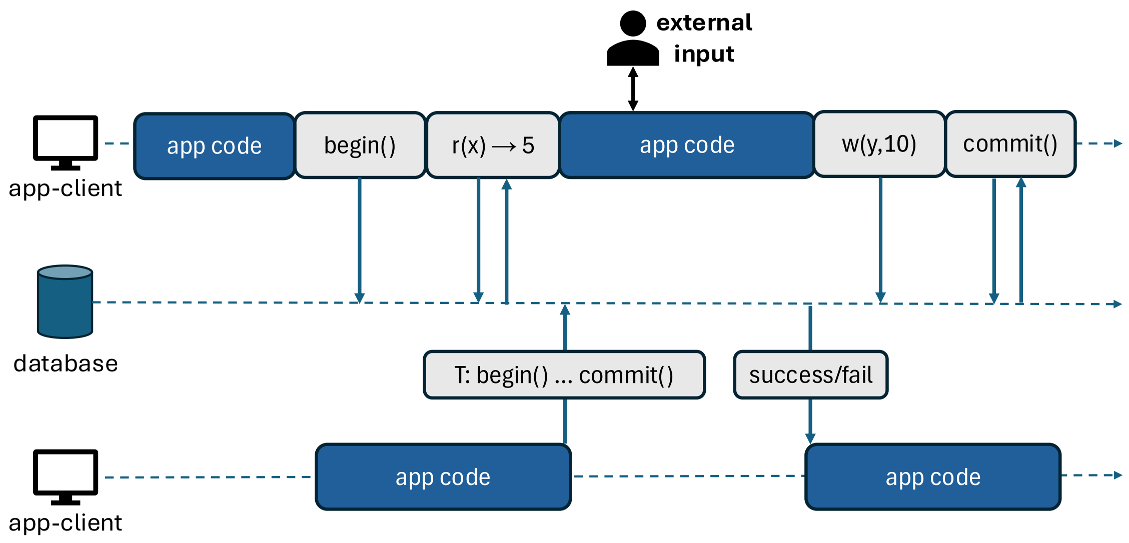

Figure 2.4: Illustration comparing an interactive transaction and one-shot transaction.

One-Shot Transactions One-shot transactions represent a simpler and more restricted model, in which the entire transaction is processed as a single, self-contained request. All operations (*e.g.*, SQL statements) are bundled together and sent to the database in a single message, with no further interaction between client and database during execution (Fig. 2.4). As the name suggests, one-shot transactions are executed in *one shot*, and preclude interaction across multiple servers during execution [95]. This model is well-suited for simple, predictable transactions, as it minimizes communication overhead by requiring only a single round-trip. However, it lacks the flexibility and expressiveness of interactive transactions and is therefore not suitable for general-purpose use.

Stored Procedures Similar to one-shot transactions, stored procedures consist of a pre-defined sequence of operations executed without interaction between the client and the database. However, unlike one-shot transactions, the transaction logic resides entirely on the server side and may support distributed execution across multiple servers [37, 89]. While stored procedures limit user flexibility, they enable efficient runtime execution, as procedures can be precompiled and optimized by the database system.

2.2 Building Distributed Databases

Databases must serve requests from many users concurrently, delivering both high throughput and low latency. To scale beyond the limitations of a single machine, systems often adopt horizontal scaling by partitioning—or *sharding*—the database contents across several machines (called *shards*).[4] Sharding enables load balancing and allows transactions that access a disjoint set of objects to execute in parallel, thereby increasing overall system capacity. However, supporting transactions that span multiple shards—*i.e.*, *distributed transactions*—requires additional coordination to preserve correctness.

Distributed Transactions To execute distributed transactions safely, the system must ensure *atomic commit*: a transaction may only commit if all involved shards deem its execution serializable and unanimously agree on the outcome. This coordination is typically handled by a dedicated *transaction manager* that coordinates a *Two-Phase Commit* (2PC) protocol (Fig. 2.5). In the first phase (*prepare*), the transaction manager collects a vote on the transaction outcome from each shard and durably logs the vote tally to survive failures. In the second phase (*commit*), the manager communicates the final decision—commit if all shards voted to commit, abort otherwise—to all involved shards. This ensures that all shards reach a consistent outcome, even in the presence of partial failures.

While sharding enables systems to scale performance efficiently, scalability alone is not enough. Many applications also require high availability—often referred to as *fault tolerance*—to ensure uninterrupted service. This is particularly critical in real-world deployments that target multiple "nines" of reliability.

[4]Some systems also perform *logical* sharding within a single machine to exploit parallelism across cores. Unless otherwise specified, this dissertation focuses on *physical* sharding across machines; the coordination involved is conceptually similar in both cases.

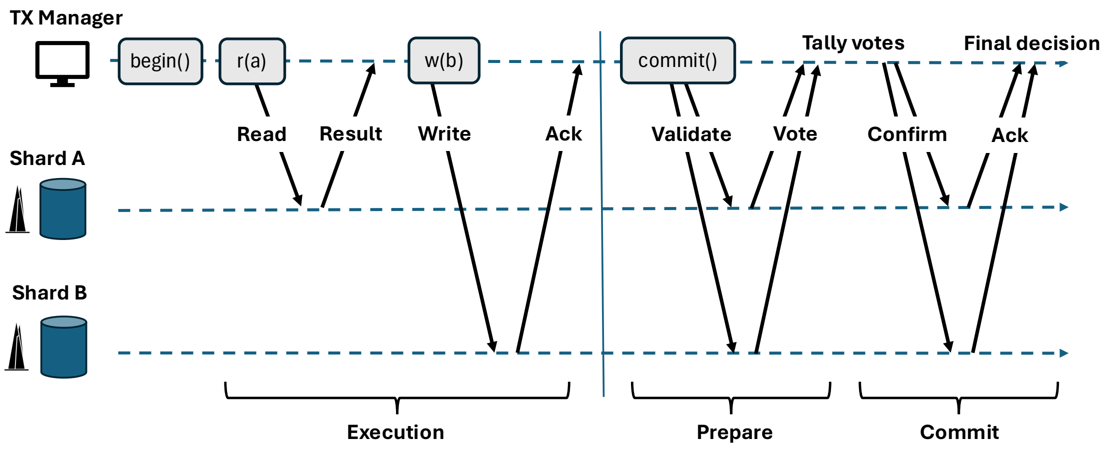

Figure 2.5: Life of a distributed transaction. The transaction accesses multiple shards during its execution and uses Two-Phase Commit (2PC) to ensure atomic commit.

Fault Tolerance To increase resilience to failures (*e.g.*, to maintain availability during shard outages) database systems often employ *replication*.[5] Instead of operating a single machine per shard, the system maintains a redundant set of *replicas*, each storing and executing the same data and transactions. We distinguish two common replication strategies: *passive* and *active* replication.

In passive replication, such as the *primary-backup* model, a single dedicated primary replica handles all client requests and asynchronously replicates its state to one or more backup replicas. If the primary fails, a backup is promoted to take over. This approach is relatively simple and introduces minimal overhead during fault-free operation, but may suffer from lag or even data loss during failover.

Active replication, by contrast, involves all replicas in processing client requests. The most prominent approach is *State Machine Replication* (SMR) [160], where replicas deterministically execute the same operations in the same order—thereby emulating the abstraction of a single, never-faulty machine. To maintain consistency, replicas first replicate client requests and agree on a global execution order—typically via a consen-

[5]Beyond fault tolerance, replication can improve scalability by distributing reads across replicas.

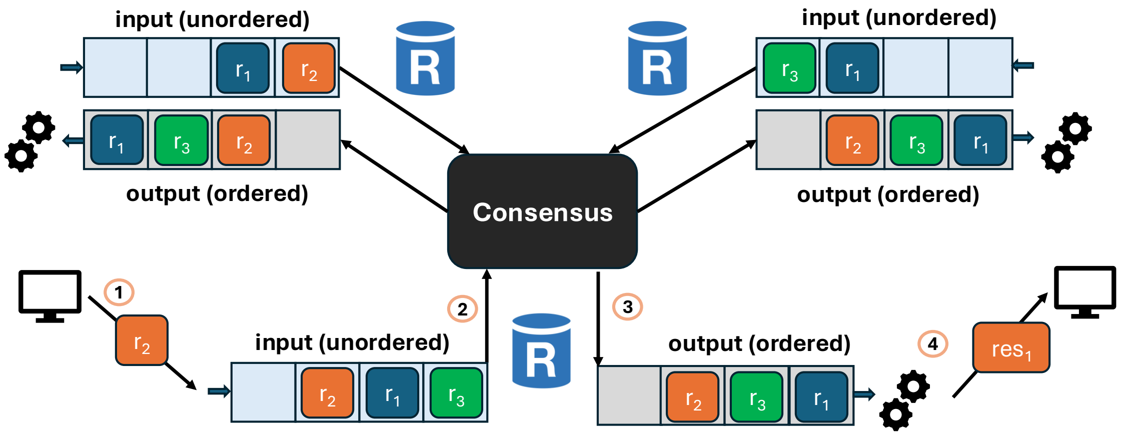

Figure 2.6: An illustration of State Machine Replication (SMR). Replicas may receive client requests in different orders, and some requests might not reach all replicas. Before execution, replicas must first replicate the requests and agree on a common execution order (consensus).

sus protocol—before executing them. While active replication introduces more coordination overhead in the fault-free case, it offers strong fault tolerance: all replicas remain consistent and fully up to date, enabling seamless recovery from failures.

Failures in distributed systems vary in nature and severity; Figure 4.11 illustrates a classic failure hierarchy. The most widely addressed failure category is that of *crash* faults, *i.e.*, failures where a machine or process stops executing abruptly and without warning. A stricter subset of crash faults, called *fail-stop*, assumes that crashes can reliably be detected, or that systems provide an advance warning before failing. Crash faults can be tolerated using either passive replication, as seen in database systems like PostgreSQL [78] or MySQL [143] that leverage primary-backup, or using active replication via SMR. Examples of the latter include distributed databases such as Google Spanner [40] or CockroachDB [175], which are built on SMR protocols such as (Multi-)Paxos [106, 107] or Raft [142].

The next class of failures is *omission* faults, where nodes may intermittently fail to send or receive messages. While omission faults are distinct in theory, they are often

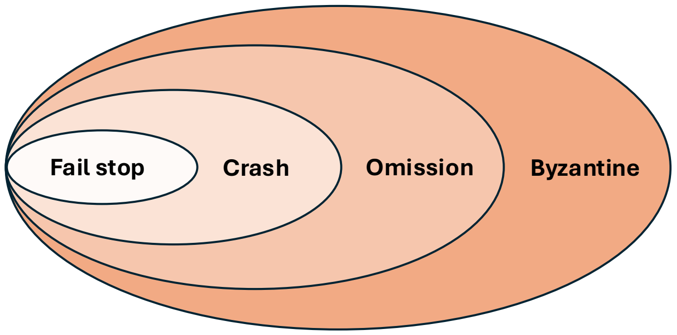

Figure 2.7: Failure hierarchy: Fail stop ⊂ Crash fault ⊂ Omission fault ⊂ Byzantine fault

modeled as crash faults in practice, with timeouts used to detect non-responsiveness.

This dissertation addresses the most general and challenging class: arbitrary failures, also known as *Byzantine* faults. In this model, faulty nodes may exhibit any behavior—including malicious actions such as sending different messages to different participants (equivocation). Because a primary replica cannot be trusted to behave correctly, only active replication is a viable solution; passive replication is fundamentally unsuitable under Byzantine assumptions.

In the following section, we provide an overview of techniques used to provide Byzantine Fault Tolerance (BFT).

2.3 Byzantine Fault Tolerance (BFT)

Byzantine Fault Tolerance (BFT) enables a distributed system to operate correctly even in the presence of arbitrarily faulty or malicious participants. This section presents an overview of BFT, its key challenges, and representative protocols.

Into the Byzantine Empire *Byzantine* failures are the most general type of failure—and the most difficult to defend against. The term "Byzantine" originates from the (in)famous Byzantine Generals Problem [111], which illustrates the challenge of achieving agreement in a distributed system where participants (*e.g.*, Byzantine army generals) may act maliciously or dishonestly. Notably, it shows that in an asynchronous network,[6] it is impossible to deterministically reach consensus with fewer than $n = 3f + 1$ participants, where at most f may be faulty. This seminal result forms the foundation of most Byzantine Fault Tolerance research.

The remainder of this section focuses specifically on Byzantine Fault Tolerant State Machine Replication (BFT SMR) protocols—*i.e.*, protocols that enable a group of participants to agree on a common request order, despite the presence of up to f faulty or malicious participants. We begin by outlining the system model and its core assumptions.

2.3.1 Model and Requirements

This dissertation focuses on BFT protocols that operate under the *partial synchrony* model [54]—that is, protocols that guarantee safety at all times (even during periods of asynchrony), but provide liveness only under favorable, synchronous conditions. In practice, these protocols use timeouts to approximate intermittent periods of synchrony.

Participants (clients or replicas) that adhere to the protocol are considered *correct*, while those that deviate are considered *faulty* (or *Byzantine*); the latter may exhibit arbitrary, potentially malicious behavior. To ensure safety, the system requires at least $n = 3f + 1$ replicas, of which at most f may be faulty at any given time. Although the system does not know which replicas are faulty, the standard model assumes that the set of faulty replicas is fixed for the duration of execution. That is, any replica that ever

[6]This impossibility result also holds in synchronous networks unless digital signatures or other cryptographic assumptions are used.

behaves incorrectly—even if only briefly—is treated as faulty from the outset. While a faulty replica may alternate between correct and incorrect behavior, the model conservatively counts it as faulty throughout. In practice, where faults can be transient and upredictable, systems approximate this assumption by dividing execution into discrete epochs using checkpoints. The set of faulty replicas can change across epochs, but the number of faulty replicas in any single epoch must remain bounded by f.

Most existing BFT protocols assume access to strong cryptographic primitives, such as unforgeable digital signatures, message authentication codes (MACs), and collision- and preimage-resistant hash functions. Digital signatures provide non-repudiation: a sender cannot credibly deny having sent a signed message, which allows BFT protocols to hold equivocating participants accountable. However, signatures are computationally expensive; excessive generation and verification can impose significant CPU overhead, especially as deployments scale to many replicas. To mitigate this overhead, some protocols use MACs to enable efficient all-to-all communication [30], while others reduce cost through linear communication patterns and signature aggregation [192].

Formal Properties The core communication primitive underlying State Machine Replication (SMR) is *Atomic Broadcast* [33]: it ensures that messages sent by participants (or *processes*) in a distributed system are delivered to all participants in the same order, even in the presence of faulty participants. To maintain consistency across participants, SMR additionally requires that message processing is deterministic.

Colloquially—and occasionally in this dissertation—the Atomic Broadcast primitive is also referred to as *consensus*. Strictly speaking, however, this is technically inaccurate, as consensus formally refers to agreement on a single decision, rather than on an ordered sequence of messages.

Concretely, Atomic Broadcast must satisfy the following four properties:

Agreeement If a correct process delivers a message *m*, then all correct processes eventually deliver *m*.

Total Order If two correct processes *p* and *q* deliver messages *m* and *m*′, then *p* delivers *m* before *m*′ if and only if *q* does the same.

Validity If a correct process broadcasts *m*, it eventually delivers *m*.[7]

Integrity A message *m* is only delivered once, and only if broadcast by its sender.

Collectively, these properties ensure that correct participants (*i*) make reliable progress (validity), (*ii*) do not process unintended operations (integrity), and (*iii*) process operations consistently (agreement and total order)—and thus never diverge in state.

Next, we outline some representative BFT SMR protocols.

2.3.2 BFT SMR Examples

We provide a brief overview of two of the most influential and widely adopted BFT SMR protocols: PBFT [30] and HotStuff [192], which serve as baselines throughout this dissertation. Representing two generations of BFT-based SMR, these protocols illustrate distinct approaches to achieving agreement in the presence of Byzantine faults, each with its own strengths and limitations. We begin by outlining the components common to both protocols.

Quorum Intersection—The Bread and Butter The foundational concept shared across all BFT SMR protocols is *quorum intersection* [125]. The core idea is straightforward: reaching agreement on a decision requires a quorum of (authenticated) replica

[7]Note that validity implicity implies termination.

votes that constitutes a super majority. This serves two critical purposes. First, this ensures that every decision is endorsed by at least one correct replica, guaranteeing that the proposed operation is well-formed and valid. Second, it ensures mutual exclusion—*i.e.*, it prevents two quorums from supporting conflicitng decisions. This property is essential to establish consistency in the presence of faulty replicas, as it thwarts equivocation attempts.

Quorum sizes depend both on the total number of replicas and the specific role of the quorum. The most common type—used to establish mutual exclusion—requires $\frac{n+f+1}{2}$ replica votes (*e.g.*, $|Q| = 2f + 1$ when $n = 3f + 1$). This ensures that any two quorums Q_1 and Q_2 intersect in at least $f + 1$ replicas, meaning that at least one correct replica is shared between them. Such a replica, by definition, cannot equivocate, and thus prevents conflicting decisions. Figure 2.8 illustrates this principle.

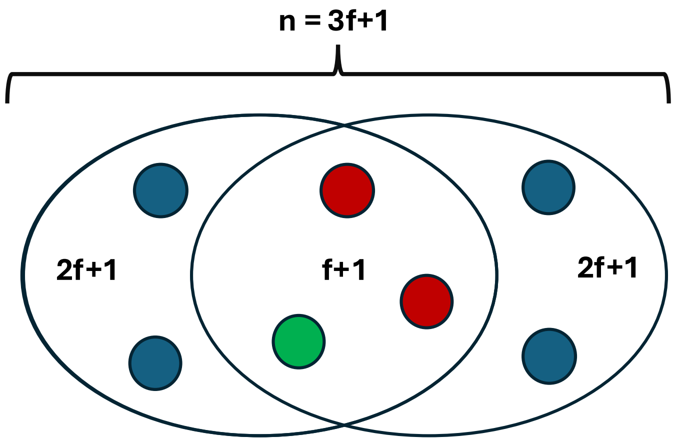

Figure 2.8: Illustration of quorum intersection with $f = 2$ and $n = 3f + 1$. Two quorums of size $2f + 1$ must intersect in at least $f + 1$ common replicas, ensuring that at least one correct (green) replica is shared between them.

Leader driven Every decision begins with a proposal. To streamline agreement, BFT SMR protocols typically designate a dedicated *leader* replica responsible for broadcasting the initial proposal. This leader acts as a sequencer to enforce *total order*, but should not be confused with a primary in passive replication. Unlike primary-backup protocols, BFT SMR requires that all correct replicas agree on a proposal before it is executed.

Leader-based BFT SMR protocols mitigate faulty leaders in two key ways. First, quorum intersection guarantees that even a faulty leader that equivocates cannot cause inconsistent decisions, preserving *agreement*. Second, to maintain liveness, protocols replace leaders that are faulty or slow. This process, known as a *view change*, transfers leadership to a new replica while preserving decisions made under the previous leader's term (or *view*). To ensure that decisions remain durable across views, most protocols require a second (or sometimes third) communication round before committing a decision. Eventually, a correct leader is elected, ensuring *validity*. Integrity is enforced at the message-level: messages include the sender's identifier and sequence number, allowing replicas to detect (and reject) duplicates.

However, leaders remain a performance bottleneck—as all requests must be sequenced and broadcast through the leader—and single point of failure, since leader failures halt all execution. Although view changes eventually restore liveness, no requests can be processed until a (possibly long) timeout expires and a correct leader is elected. Leaders also raise fairness concerns: a malicious leader can, for example, selectively censor or reorder requests for personal gain (*e.g.*, to carry out a front-running attack).

PBFT

Practical Byzantine Fault Tolerance [30]—commonly referred to as PBFT—is the most famous and widely adopted BFT SMR protocol. Its design marked a paradigm shift in buildig cost-efficient, low-latency consensus systems, and it remains the benchmark for BFT SMR to this day.

Common Case PBFT is a leader-based protocol that proceeds in three main phases: *propose*,[8] *prepare*, and *commit*. Figure 2.9 illustrates the agreement process.

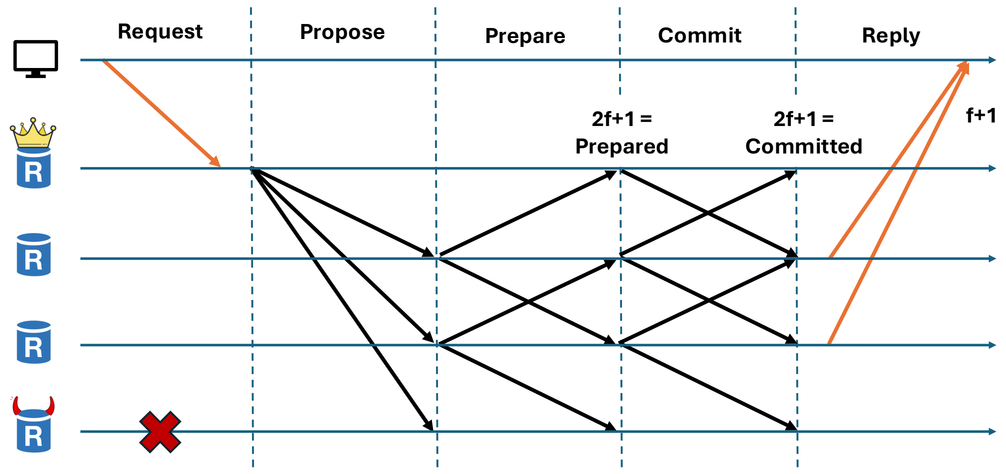

Figure 2.9: An overview of common case processing in PBFT. The fourth replica is faulty and does not participate.

In the *propose* phase, the leader broadcasts a request (*e.g.*, the next operation to execute) to the other replicas. In the prepare phase, replicas exchange confirmation messages all-to-all. A replica considers a proposal *prepared* upon receiving $\frac{n+f+1}{2} = 2f + 1$ PREPARE messages (including its own). By quorum intersection, no conflicting proposal can be accepted (property P1), ensuring that all correct replicas agree on the total order of requests within the current leader's view. During the commit phase, replicas broadcast their accepted decision once more to ensure durability. Finally, if a replica receives $2f + 1$ matching COMMIT votes, it considers the ordering decision final and executes the proposed request. It follows that if any correct replica commits to a decision, at least $f + 1$ correct replicas must have considered the decision prepared (property P2).

In the fault-free, synchronous case, a leader proposal requires three message delays to be committed. The end-to-end latency for a client request is therefore (at least) five message delays—one to send the request, (at least) three for agreement, and one to

[8]This phase is called *pre-prepare* in the original PBFT paper [30]; we use *propose* here for clarity.

receive the result. Note that a client accepts a result only after receiving $f + 1$ matching replies, enough to assert that it was endorsed by at least one correct replica.

View Change If a correct replica fails to reach a decision within a predefined timeout—either because the leader equivocated, or because the leader or other replicas were slow—it initiates a *view-change* to replace the current leader. The replica halts participation in the current view and sends a NEW-VIEW message to the leader of the next view. This message includes the replica's current ledger state and the set of prepared proposals, along with evidence of $2f + 1$ signed PREPARE messages.

The new leader considers itself elected once it receives $2f + 1$ NEW-VIEW messages. To ensure safety, it must recover any decision that could have been committed—specifically, any proposal that received at least $f + 1$ PREPARE votes from correct replicas (per P2). By quorum intersection, any quorum of size $2f + 1$ must contain at least one NEW-VIEW message containing the prepared decision and valid endorsements. According to P1, at most one valid prepare decision can exist in any view, allowing the leader to safely recover it. If the leader receives valid but conflicting prepare decisions from different views, it resolves the conflict by choosing the decision from the highest view—*i.e.*, it breaks ties by preferring more recent views.

Optimizations To amortize agreement costs, PBFT batches multiple client requests into a single proposal. Additionally, the leader can drive agreement for succeccive proposals—*i.e.*, those assigned to consecutive ledger positions or *slots*—in parallel. To reduce cryptographic overhead, PBFT uses digital signatures only for new-view and view chage messages, relying on lightweight message authentication codes (MACs) for all other communication. Finally, to minimize latency, PBFT offers an optional optimization that allows a client to commit upon receiving $2f + 1$ matching tentative results following the *prepare* phase, reducing end-to-end latency to four message delays. However, this is rarely implemented as it requires replicas to speculatively execute operations and support state rollbacks if the tentative decision is not preserved across views.

Many subsequent works have extended and improved PBFT. Zyzzyva [101] introduces a fast path that allows clients to commit in only three end-to-end message delays under fault-free conditions; SBFT [80] adopts a similar design but leverages signature aggregation [29, 45, 163] and linear communication patterns to improve scalability. Aardvark [35] increases robustness by rotating leaders proactively, while UpRight [34] reduces BFT overheads by explicitly distinguishing Byzantine and crash failures. Autobahn [73] scales throughput and improves resilience to intermittent failures by decoupling data dissemination and agreement—allowing every replica to act as proposers and streamlining recovery.

HotStuff

PBFT offers low latency but relies on all-to-all communication, leading to quadratic communication complexity in the common case and cubic complexity for view changes. HotStuff [192] aims to reduce communication complexity by unifying the view change with the common case and leveraging symmetry in the voting process. This design simplifies implementation and achieves linear communication complexity—when using signature aggregation techniques—though at the cost of increased latency.

HotStuff, like PBFT, is leader-based and follows a similar multiphase protocol, but introduces an additional *pre-commit* phase to the common case execution, effectively integrating the view-change into normal operation. This phase is necessary to achieve linear communication complexity though it is not strictly required for safety [192]. HotStuff proceeds through four symmetric phases: *propose*, *prepare*, *pre-commit*, and *com-*

mit. Each phase employs a linear communication pattern where the leader collects a set votes—forming a *quorum certificate* (QC)—and then broadcasts this certificate to drive the next phase. Figure 2.10 illustrates this agreement process.

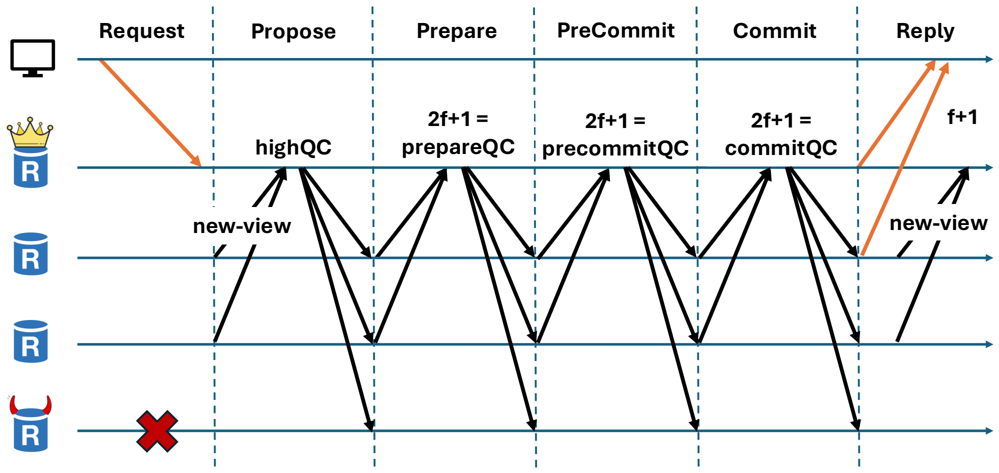

Figure 2.10: An overview of common case processing in (non-chained) HotStuff. The fourth replica is faulty and does not participate.

Inspired by blockchain designs, HotStuff adopts a *block-based* consensus approach. Leader proposals are structured as blocks, each containing a cryptographic hash of its preceeding block, thereby forming a hash-chain that ensures immutability.

Basic HotStuff In HotStuff, each proposal corresponds to a new view. The leader initiates agreement by proposing a new block (*propose* phase). In the *prepare* phase, the leader collects $\frac{n+f+1}{2} = 2f + 1$ votes for its proposal and assembles (and forwards) a PREPAREQC, ensuring mutual exclusion and preventing equivocation. This process repeats in the *pre-commit* phase, where replicas acknowledge receipt of the PREPAREQC and the leader forms and disseminate a PRECOMMITQC. Finally, during the *commit* phase, the leader gathers and forwards votes that constitute a COMMITQC. Once replicas receive the COMMITQC, they commit the proposed block, advance to the next view, and initiate a new round of agreement by sending a NEW-VIEW message to the next leader. If a leader is slow or faulty, replicas send NEW-VIEW messages after a timeout to trigger a view change and maintain liveness.

In the fault-free synchronous case, committing a proposal requires seven message delays. Including client communication, the end-to-end latency for a request is (at least) 9 message delays. Similar to PBFT, a client only accepts a result upon receiving $f + 1$ matching replies, ensuring endorsement by at least one correct replica.

HotStuff integrates the view change directly into the common case protocol via subtle voting rules. At the start of each *propose* phase (a new view), the leader must gather $2f + 1$ NEW-VIEW messages before proposing a block. To preserve safety, the leader extends the block with the highest PREPAREQC (HIGHQC), ensuring the committed chain never forks. No conflicting proposal from a lower view (not included in the hash chain of HIGHQC) can have committed. Replicas enforce locking rules to maintain consistency: each replica locks on the highest PRECOMMITQC it has observed and only accepts a new proposal if either (*i*) the proposal extends the locked block, or (*ii*) it belongs to a higher view, guaranteeing that the locked block could not have committed.

Optimizations At first glance, HotStuff still incurs quadratic communication overhead. However, it can achieve linear complexity through signature aggregation [29, 45, 163], which compresses each quorum certificate (QC) into a single aggregated signature—reducing the cost of forwarding a QC to constant. This optimization, however, is rarely used in practice, as signature aggregation is only resource-efficient in large replication groups [80] (*e.g.*, $n \geq 100$).

To further amortize cryptographic costs and reduce latency between proposals, HotStuff pipelines multiple agreement rounds by leveraging the symmetry of its voting phases. Each phase can propose a new block, and a single QC can simultaneously serve as the PREPAREQC, PRECOMMITQC, and COMMITQC for consecutive proposals. A block is considered committed only when it is certified by three consecutive QCs. Pipelining can be combined with eager leader rotation [35] by treating each voting phase as a new view. This approach promotes fairness by distributing proposal responsibilities across replicas. However, it introduces new liveness vulnerabilites, as successfull committment now requires three successive correct leaders [74].

Recent follow-up work has sought to reduce HotStuff's latency by eliminating the *pre-commit* phase—either by sacrificing linear communicaton complexity [67, 74, 91], extending view changes [168, 169], or leveraging novel cryptography techniques [72]. Neverthleess, agreement latency remains high: committing a leader's proposal still requires at least five message delays, and at end-to-end client confirmation requires at least seven.

2.4 Distributed Databases—Putting it All Together

Distributed databases consist of many interdependent components. To achieve horizontal scalability, they shard data; to tolerate faults, they replicate it; and to preserve data integrity, they orchestrate serializable transactions. A common architectural pattern follows a layered approach, as illustrated in Figure 2.11. At the base lies a replication layer that employs State Machine Replication (SMR) to ensure consistency, making each shard appear as a single, fault-tolerant entity. On top of this, each shard runs a concurrency control (CC) protocol to enforce serializability. Finally, a Two-Phase Commit (2PC) protocol coordinates atomic transaction commit across shards to guarantee global correctness.

This architecture is conceptually simple and promotes modularity, allowing each protocol to be designed independently. The database may be shared in various

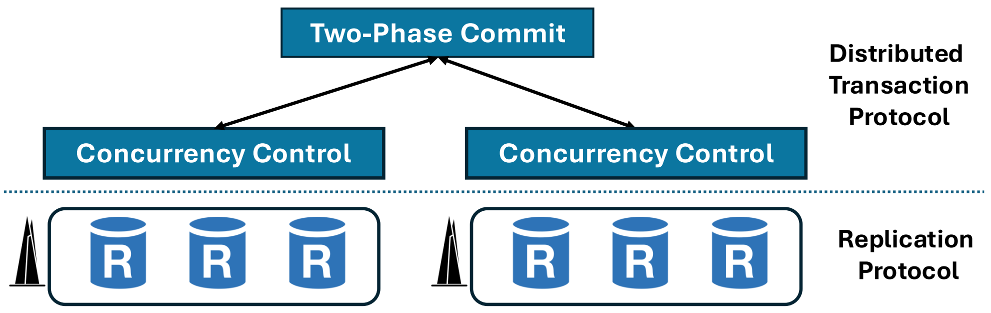

Figure 2.11: A common architecture for distributed databases. A distributed transaction protocol—consisting of Two-Phase Commit and Concurrency Control—is layered atop a strongly consistent replication protocol. Reproduced from Zhang et al. [195]

ways—or not at all—depending on system requirements. Internally, shards (or the unsharded database) can be replicated using different protocols: for example, (Multi-)Paxos [106, 107] or Raft [142] in environments concerned only with crash faults, or PBFT [30] or HotStuff [192] when Byzantine fault tolerance is required. Likewise, systems vary in their choice of concurrency control, with some adopting pessimistic approaches like Two-Phase Locking (2PL), while others favor optimistic approaches such as Multi-Version Concurrency Control (MVCC).

Examples Several widely used production systems follow this classic layered architecture. For example, Google Spanner [40] combines 2PL for read-write transactions with MVCC for snapshot reads. It uses (Multi-)Paxos for replication and employs 2PC to ensure atomicity for distributed transactions. Modern cloud-native databases like CockroachDB [37, 175], YugabyteDB [89] and TiDB [86, 153] adopt a similar architectural approach, but build atop Raft [142] for replication. They layer MVCC and 2PC atop Raft to provide strong consistency guarantees and scalable transaction processing.

Shortcomings of Layered Designs Modularity, while convenient, comes at a cost: each layer enforces consistently independently—sometimes redundantly—leading to inefficiencies [195, 196]. The concurrency control layer enforces serializability, ensuring that transactions appear to execute in a total order. Meanwhile, the replication layer imposes its own global ordering on all transaction operations (within a shard) to maintain consistency across replicas. Unfortunately, this redundant enforcement of ordering strips away much of the natural parallelism inherent to transaction processing, significantly constraining overall system performance.

Ordering can also incur substantial latency, especially in Byzantine settings. For instance, PBFT requires at least four message delays before a request can be processed; one for the transaction coordinatior (*e.g.*, the client) to submit the request, and at least three more for the replicas to reach agreement and commit. Additionally, ordering depends on a leader, which can become a throughput bottleneck. Layering 2PC atop replication further amplifies this cost, since each round of the 2PC protocol must itself be totally ordered within every involved shard.

In practice, these latency overheads make interactive transactions impractical in Byzantine fault tolerant systems. As a result, most existing BFT systems restrict their transaction model to one-shot transactions or pre-defined stored procedures—often called *smart contracts* in blockchain contexts [7, 8, 16, 198].

Integrating Layers Recent work observes that not all operations require total ordering and investigates the minimal ordering requirements necessary to ensure consistency in distributed databases [148, 195, 196].

For example, under optimistic concurrency control, reads do not modify replica state and therefore need not be replicated or ordered. In crash fault tolerant settings, this allows reads to be served from a single—possibly local—replica, improving latency. In contrast, locking-based approaches require total ordering to maintain consistency, making them less favorable performance-wise. We note that total ordering does not necessarily imply replication: locking-based systems such as Google Spanner register reads at a leader replica for ordering but do not replicate them. However, this still introduces additional coordination and at least one round-trip delay.

TAPIR [195] explores the design of a crash fault tolerant distributed database that provides strong transaction-level consistency (serializability) while relaxing consistency requirements at the replication layer. Specifically, TAPIR introduces a novel primitive called *inconsistent replication*, which allows replicas to execute transactions in any order. It distinguishes between *inconsistent operations*, which can execute out of order without requiring agreement but require replication (*e.g.*, writes in a multi-versioned system), and *consensus operations*, which also execute out of order but require agreement on their result (*e.g.*, transaction validation checks). Notably, TAPIR avoids replicating reads altogether, relying instead on concurrency control to ensure end-to-end consistency.

Several systems have built upon this design, most notably Meerkat [174], which accelerates TAPIR by minimizing cross-core coordination within replicas, and Morty [28], which combines inconsitent replication and dynamic transaction re-execution to achieve high performance even in wide-area deployments.

This dissertation Despite significant progress in scaling crash fault tolerant (CFT) database systems, Byzantine fault tolerant (BFT) systems today continue to rely on total ordering. As a result, they either offer performance that falls short of their CFT counterparts or impose restrictive transaction models—making them ill-suited as drop-in replacements for existing CFT systems or as general-purpose platforms for distributed applications.

This dissertation aims to design and build BFT distributed databases that not only compete with CFT systems in performance but also support rich, expressive transaction interfaces—such as interactive transactions—that developers prefer. More specifically, it investigates whether the cross-layer design principles proven effective in CFT systems can be extended to the Byzantine setting, and whether doing so can yield practical and efficient systems. Realizing this goal is not straighftoward. For example, even a simple read poses a challenge: unlike in CFT settings, a single replica cannot be trusted, so some form of replication is inherently required. Similarly, optimistic concurrency control is typically client-driven, yet Byzantine clients cannot be trusted to uphold correctness or liveness—for instance, they may ignore transaction conflicts or intentionally roll back their writes to cause cascading transaction aborts.

In the chapters that follow, we explore how to build *database-first* BFT systems that avoid total ordering and coordinate only when necessary, opening a new design space for efficient and expressive BFT transaction processing.

CHAPTER 3

BASIL: BREAKING UP BFT WITH ACID (TRANSACTIONS)

The previous chapter established the background on distributed transaction systems and Byzantine fault tolerance (BFT). This chapter presents Basil[1]—a leaderless and shardable transactional key-value store that scales the abstraction of a BFT totally ordered log while simultaneously increasing robustness.

The context Byzantine fault tolerant (BFT) systems enable safe online data sharing among mutually distrustful parties by guaranteeing correctness even in the presence of malicious (Byzantine) actors. These platforms open exciting opportunities across various domains, including healthcare [118], financial services [13, 58, 87], and supply chain management [47, 186]. At the core of these services are BFT replicated state machines [30, 35, 80, 101, 192] and permissioned blockchains [7, 8, 16, 27, 71, 98], which ensure that mutually distrustful parties agree on the same totally ordered log of operations.

The abstraction of a totally ordered log is appealingly simple. A *scalable* totally ordered log, however, is not only hard to implement (processing all requests sequentially can become a bottleneck), but also often unnecessary. Most distributed applications primarily consist of logically concurrent operations; supply chains for instance, despite their name, are actually complex networks of independent transactions.

Some BFT systems use sharding to try to tap into this parallelism. Transactions that access disjoint shards can execute concurrently, but operations within each shard are still totally ordered. Transactions involving multiple shards are instead executed by running cross-shard atomic commit protocols, which are layered above these totally ordered

[1]Despite (or because of) his deeply *Fawlty* character, Basil managed to rise first from peasant to Byzantine emperor (867-886) and then to hotel owner (1975-1979).

shards [7, 98, 145, 146, 194]. The drawbacks of systems that adopt this architecture are known: (*i*) they pay the performance penalty of redundant coordination—both across shards (to commit distributed transactions) and among the replicas within each shard (to totally order in-shard operations) [138, 195, 196]; (*ii*) within each shard, they give a leader replica undue control over the total order ultimately agreed upon, raising fairness concerns [85, 192, 197]; (*iii*) and often they restrict the expressiveness of the transactions they support [145, 146] by requiring that their read and write set be known in advance.

Our proposal In this dissertation, we advocate a more principled, performant, and expressive approach to supporting the abstraction of a totally ordered log at the core of all permissioned blockchain systems. We make our own the lesson of distributed databases, which successfully leverage generic, interactive transactions to implement the abstraction of a sequential, general-purpose log. These systems specifically design highly concurrent protocols that are *equivalent* to a serial schedule [22, 147]. Byzantine data processing systems need be no different: rather than aiming to sequence all operations, they should decouple the *abstraction* of a totally ordered sequence of transactions from its *implementation*. Thus, we flip the conventional approach: instead of building database-like transactions on top of a sharded, totally ordered BFT log, we directly build out this log abstraction above a partially-ordered distributed database, where total order is demanded only for conflicting operations.

Basil, at glance To this effect, we design Basil, a serializable BFT key-value store that implements the abstraction of a trusted shared log, whose novel design addresses each of the drawbacks of traditional BFT systems: (*i*) it borrows databases' ability to leverage concurrency control to support highly concurrent but serializable transactions, thereby adding parallelism to the log; (*ii*) it sidesteps concerns about the fairness of leader-based systems by giving clients the responsibility of driving the execution of their own transactions; (*iii*) it eliminates redundant coordination by integrating distributed commit with replication [138, 195], so that, in the absence of faults and contention, transactions can return to clients in a single round trip; and (*iv*) it improves the programming API, offering support for general interactive transactions that do not require a-priori knowledge of reads and writes.

We lay the foundations for Basil by introducing two complementary notions of correctness. *Byzantine isolation* focuses on safety: it ensures that correct clients observe a state of the database that could have been produced by correct clients alone. *Byzantine independence* instead safeguards liveness: it limits the influence of Byzantine actors in determining whether a transaction commits or aborts. To help enforce these two notions, and disentangle correct clients from the maneuvering of Byzantine actors, Basil's design follows the principle of *independent operability*: it enforces safety and liveness through mechanisms that operate on a per-client and per-transaction basis. Thus, Basil avoids mechanisms that enforce isolation through pessimistic locks (which would allow a Byzantine lock holder to prevent the progress of other transactions), adopting instead an optimistic approach to concurrency control.

Embracing optimism in a Byzantine setting comes with its own risks. Optimistic concurrency control (OCC) protocols [20, 104, 156, 187, 195] are intrinsically vulnerable to aborting transactions if they interleave unfavorably during validation, and Byzantine faults can compound this vulnerability. Byzantine actors may, for instance, intentionally return stale data, or collude to sabotage the commit chances of correct clients' transactions. Consider multiversioned timestamp ordering (MVTSO) [20, 156], which allows writes to become visible to other operations before a transaction commits. While this choice helps reduce abort rates for contended workloads, it can cause transactions to stall on uncommitted operations.

Basil's ethos of independent operability is key to mitigating this issue. The system implements a variant of MVTSO that prevents Byzantine participants from unilaterally aborting correct clients' transactions and includes a novel *fallback* mechanism that empowers clients to finish pending transactions issued by others, while preventing Byzantine actors from dictating their outcome. Importantly, this fallback is a per-transaction recovery mechanism: thus, unlike traditional BFT view-changes, which completely suspend the normal processing of all operations, it can take place without blocking non-conflicting transactions.

The benefits Our results are promising: on TPC-C [182], Smallbank [50], and a Retwis-based workload [113, 195], Basil's throughput is 3.5-5x higher than layering distributed commit over totally ordered shards running BFT-SMaRt, a state-of-the-art PBFT implementation [23] and HotStuff [192] (Facebook Diem's core consensus protocol [16]). BFT's cryptographic demands, however, still cause Basil to be 2-4 times slower than TAPIR, a recent non-Byzantine distributed database [195]. In the presence of Byzantine clients, Basil's performance degrades gracefully: with 30% Byzantine clients, the throughput of Basil's correct clients drops by less than 25% in the worst-case.

In summary, this chapter makes the following three contributions:

1. It introduces the complementary correctness notions of Byzantine isolation and Byzantine independence.
2. It presents novel concurrency control, agreement, and fallback protocols that balance the desire for high-throughput in the common case with resilience to Byzantine attacks.
3. It describes Basil, a BFT database that guarantees Byz-serializability while preserving Byzantine independence. Basil supports interactive transactions, is leaderless, and achieves linear communication complexity.

Roadmap This chapter is structured as follows. Section 3.1 formalizes Basil's correctness guarantees. Section 3.2 outlines Basil's architecture. We detail our concurrency control protocols and recovery protocols in Section 3.3 and Section 3.4; Section 3.5 proves their correctness. We evaluate Basil in Section 3.6, discuss related work in Section 3.8, and conclude in Section 3.9.

3.1 Model and Definitions

We introduce the complementary and system-independent notions of Byzantine isolation and Byzantine independence, which, jointly formalize the degree to which a Byzantine actor can affect transaction progress and safety.

3.1.1 System Model

Basil inherits the standard assumptions of prior BFT work. A participant is considered correct if it adheres to the protocol specification, and faulty otherwise. Faulty clients and replicas may deviate arbitrarily from their specification; a strong but static adversary can coordinate their actions but cannot break standard cryptographic primitives. A shard contains a partition of the data in the system.

We assume that the number of faulty replicas within a shard does not exceed a threshold f and that an arbitrary number of clients may be faulty; we make no further assumption about the pattern of failures across shards. We assume that applications authenticate clients and can subsequently audit their actions.

Similar to other BFT systems [27, 30, 35, 59, 101], Basil makes no synchrony assumption for safety but for liveness [59] depends on partial synchrony [54].

Basil also inherits some of the limitations of prior BFT systems: it cannot prevent authenticated Byzantine clients, who otherwise follow the protocol, from overwriting correct clients' data. Basil does not enforce whether authenticated clients adhere the intended application semantics. A Byzantine client, for example, might violate the semantics of payment transaction and "double-spend" funds. We assume that authenticated clients can be held accountable by an external service audits committed transactions after the fact.

We additionally assume that, collectively, Byzantine and correct clients have similar processing capabilities, and thus Byzantine clients cannot cause a denial of service attack by flooding the system.

3.1.2 System Properties

To express Basil's correctness guarantees, we introduce the notion of *Byzantine isolation*. Database isolation (serializability, snapshot isolation, etc.) traditionally regulates the interaction between concurrently executing transactions; Byzantine isolation ensures that, even though Byzantine actors may choose to violate ACID semantics for themselves, the state observed by correct clients will always be ACID compliant.

We start from the standard notions of transactions and histories introduced by Bernstein et al. [21]. We summarize them here and defer a more formal treatment to our correctness proofs (§ 3.5). A transaction T contains a sequence of read and write operations terminating with a commit or an abort. A history H is a partial order of operations representing the interleaving of concurrently executing transactions, such that all conflicting operations are ordered with respect to one another. Additionally, let C be the set of all clients in the system; $Crct \subseteq C$ be the set of all correct clients; and $Byz \subseteq C$ be the set of all Byzantine clients. A projection $H|_{\mathscr{C}}$ is the subset of the partial order of operations in H that were issued by the set of clients $\mathscr{C}$. We further adopt standard definitions of database isolation: a history satisfies an isolation level I if the set of operation interleavings in H is allowed by I. Drawing from the notions of BFT linearizability [122] and view serializability [21], we then define the following properties:

Definition (Legitimate History) *History H is legitimate if it was generated by correct participants, i.e., $H = H_{Crct}$.*

Definition (Correct-View Equivalent) *History H is correct-view equivalent to a history H' if all operation results, commit decisions, and final object values in $H|_{Crct}$ match those in H'.*

Definition (Byz-I) *Given an isolation level I, a history H is Byz-I if there exists a legitimate history H' such that H is correct-view equivalent to H' and H' satisfies I.*

This definition is not Basil-specific, but captures what it means, for any Byzantine-tolerant database, to enforce the guarantees offered by a given isolation level I. Informally, it requires that the states observed by correct clients be explainable by a history that satisfies I and involves only correct participants. It intentionally makes no assumptions on the states that Byzantine clients choose to observe.

Basil specifically guarantees Byz-serializability: correct clients will observe a sequence of states that is consistent with a sequential execution of concurrent transactions. This is a strong safety guarantee, but it does not enforce application progress; a Byz-serializable system could still allow Byzantine actors to systematically abort all transactions. We thus define the notion of *Byzantine independence*, a general system property that bounds the influence of Byzantine participants on the outcomes of correct clients' operations.

Definition (Byzantine Independence) *For every operation o issued by a correct client c, no group of participants containing solely Byzantine actors can unilaterally dictate the result of o.*

In a context where clients issue transaction operations, Byzantine independence implies, for instance, that Byzantine actors cannot collude to single-handedly abort a correct client's transaction. This is a challenging property to enforce. It cannot be attained in a leader-based system: if the leader and a client are both Byzantine, they can collude to prevent a transaction from committing by strategically generating conflicting requests. In contrast, Basil can enforce Byzantine independence as long as Byzantine actors do not have full control of the network, a requirement that is in any case a precondition for any BFT protocol that relies on partial synchrony [59, 132]. We prove in Section 3.5 that:

Theorem 1 *Basil maintains Byz-serializability.*

Theorem 2 *Basil maintains Byzantine independence in the absence of a network adversary.*

Basil is designed for settings where Byzantine attacks can occur, but are infrequent, consistent with the prevalent assumption for permissioned blockchains today; namely, that to maintain standing in a permissioned system, clients are unlikely to engage in actively detectable Byzantine behavior [81, 82] and, if they cannot break safety undetected, it is preferable for them to be live [124]. We design Basil to be particularly efficient during gracious executions [35] (*i.e.*, synchronous and fault-free) while bounding overheads when misbehavior does occur. In particular, we design aggressive concurrency control mechanisms that maximize common case performance by optimistically exposing uncommitted operations, but ensure that these protocols preserve independent operability, so that Basil can guarantee continued progress under Byzantine attacks [35]. We confirm this experimentally in Section 3.6.

3.2 Overview

Basil is a transactional key-value store designed to be scalable and leaderless. Our architecture reflects this ethos.

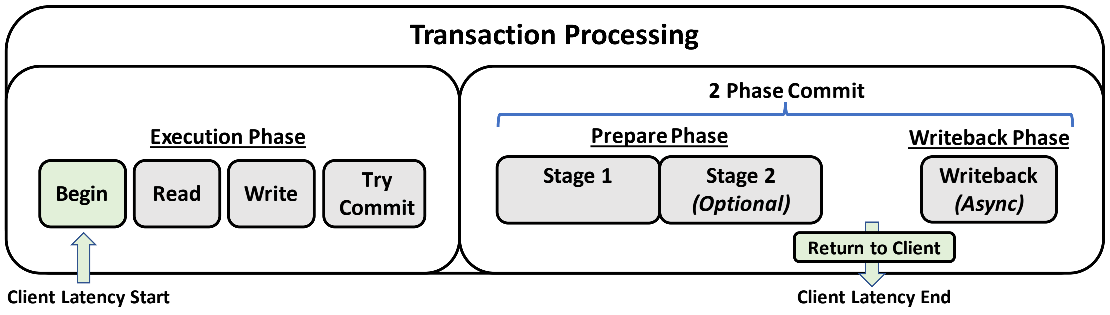

Figure 3.1: Basil Transaction Processing Overview

Transaction Processing Transaction processing is driven by clients (avoiding costly all-to-all communications amongst replicas) and consists of three phases (Figure 3.1).

First, in an *Execution phase*, clients execute individual transactional operations. As is standard in optimistic databases, reads are submitted to remote replicas while writes are buffered locally. Basil supports interactive and cross-shard transactions: clients can issue new operations based on the results of past operations to any shard in the system.

In a second *Prepare phase*, individual shards are asked to vote on whether committing the transaction would violate serializability. For performance, Basil allows individual replicas within a shard to process such requests out of order. For progress, Basil must ensure that Byzantine replicas cannot cause transactions to abort arbitrarily.

Finally, the client aggregates each shard vote to determine the outcome of the transaction, notifies the application of the final decision, and forwards the decision to the participating replicas in an asynchronous *Writeback phase*. Importantly, the decision of whether each transaction commits or aborts must be preserved across both benign and Byzantine failures. We describe the protocol in detail in Section 3.3.

Transaction Recovery A Byzantine actor could begin executing a transaction, run the prepare phase, but intentionally never reveal its decision. Such behavior could prevent other transactions from making progress. Basil thus implements a *fallback recovery mechanism* (§ 3.4) that can terminate stalled transactions while preserving Byz-serializability. This protocol, in the common case, allows clients to terminate stalled transactions in a single additional round-trip.

Replication Basil uses $n = 5f + 1$ replicas for each shard. This choice allows Basil to (*i*) preserve Byzantine independence (*ii*) commit transactions in a single round-trip in the absence of contention, and (*iii*) reduce the message complexity of transaction recovery by a factor of n—all features which would be unattainable with a lower replication factor. We expand on this further in Sections 3.3.6, 3.4, and 3.7.6.

3.3 Transaction Execution

Basil takes as its starting point MVTSO [20], an optimistic multiversioned concurrency control, and modifies it in three ways: (*i*) in the spirit of independent operability, it has clients drive the protocol execution; (*ii*) it merges concurrency control with replication; and finally (*iii*) it hardens the protocol against Byzantine attacks to guarantee

Byz-serializability while preserving Byzantine independence.

Traditional MVTSO works as follows. A transaction T is assigned (usually by a transaction manager or scheduler) a unique timestamp ts_T that determines its serialization order. As MVTSO is multiversioned, writes in T create new versions of the objects they touch, tagged with ts_T. Reads instead return the version of the read object with the largest timestamp smaller than ts_T and update that object's *read timestamp* (RTS) to ts_T. Read timestamps are key to preserving serializability: to guarantee that no read will miss a write from a transaction that precedes it in the serialization order, MVTSO aborts all writes to an object from transactions whose timestamp is lower than the object's RTS.

MVTSO is an optimistic protocol, and, as such, much of its performance depends on whether its optimistic assumptions are met. For example, it uses timestamps to assign transactions a serialization order a-priori, under the assumption that those timestamps will not be manipulated; further, it allows read operations to become *dependent* on values written by ongoing transactions under the expectation that they will commit. This sunny disposition can make MVTSO particularly susceptible to Byzantine attacks. Byzantine clients could use artificially high timestamps to make lower-timestamped transactions less likely to commit; or they could simply start transactions that write to large numbers of keys and never commit them: any transaction dependent on those writes would be blocked too. At the same time, by blocking on dependencies (rather than summarily aborting, as OCC would do) MVTSO leaves open the possibility that blocked transactions may be rescued and brought to commit. In the remainder of this section, we describe how Basil, capitalizing on this possibility, modifies MVTSO to harden it against Byzantine faults.

3.3.1 Execution Phase

Begin() A client begins a transaction T by optimistically choosing a timestamp $ts :=$ *(Time, ClientID, ClientSeqNo)* that defines a total serialization order across all clients. Allowing clients to choose their own timestamps removes the need for a centralized scheduler, but makes it possible for Byzantine clients to create transactions with arbitrarily high timestamps: objects read by those transactions would cause conflicting transactions with lower timestamps to abort. To defend against this attack, replicas accept transaction operations if and only if their timestamp is no greater than $R_{Time} + \delta$, where R_{Time} is the replica's own local clock. Neither Basil's safety nor its liveness depend on the specific value of δ, though a well-chosen value will improve the system's throughput. In practice, we choose δ based on the skew of NTP's clock and average client-replica ping latency. We discuss in Section 3.7 some alternative mechanisms to curb misbehavior.

Write(key,value) Writes from uncommitted transactions raise a dilemma. Making them readable empowers Byzantine clients to stall all transactions that come to depend on them. Waiting to disclose them only when the transaction commits, however, increases the likelihood that concurrent transactions will abort. We adopt a middle ground: we buffer writes locally at clietns until the transaction has finished execution, and make them visible during the protocol's Prepare phase (we call such writes *prepared*). This approach allows us to preserve much of the performance benefits of early write disclosure while enforcing independent operability (§ 3.3.2).

Read(key) In traditional MVTSO, a read for transaction T returns the version of the read object with the highest timestamp smaller than ts_T. When replicas process requests independently, this guarantee no longer holds, as the write with the largest timestamp smaller than ts_T may have been made visible at replica R, but not yet at R': reading from the latter may result in a stale value. Hence, to ensure serializability, transactions in Basil go through a concurrency control check at each replica as part of their Prepare phase (§ 3.3.2). Further care is required, as Byzantine replicas could intentionally return stale (or imaginary!) values that would cause transactions to abort, violating Byzantine independence. These considerations lead us to the following read logic:

1: C → R: Client C sends read request to replicas.

C sends an authenticated read request $m := \langle \textsc{read}, key, ts_T \rangle$ to at least $2f + 1$ replicas in the shard S that is responsible for *key*.

2: R → C: Replica processes client read and replies.

Each replica R verifies that the request's timestamp is smaller than $R_{Time} + \delta$. If not, it ignores the request; otherwise, it updates *key*'s RTS to ts_T. Unlike traditional MVTSO, the RTS in Basil is not required for safety but serves as an optimization that improves the commit chances of readers (§ 3.3.3). Basil may opt to evict clients with a history of reading keys but never committing the transaction.

Next, R returns a signed message $\langle Committed, Prepared \rangle_{\sigma_R}$ that contains, respectively, the latest committed and prepared versions of *key* at R with timestamps smaller than ts_T. $Committed \equiv (version, \textsc{c-cert})$ includes a *commit certificate* c-cert (§ 3.3.3) proving that *version* has committed, while $Prepared \equiv (version, id_{T'})$ includes a digest identifier for the transaction T' that created *version*.

3: C ← R: Client receives read replies.

A client waits for at least $f + 1$ replies (to ensure that at least one comes from a correct replica) and chooses the *highest-timestamped* version that is *valid*. For committed versions, the criterion for validity is straightforward: a committed version must contain a valid c-cert, proving that the write is indeed committed. For prepared versions instead, we require that the same version be returned by at least $f + 1$ replicas. This ensures that at least one correct replica (*i*) has applied the write and expects it to commit, and (*ii*) is in possession of the writer's full transaction object and can aid in Basil's cooperative *fallback* protocol (§ 3.4).

Both the validity and timestamp requirement are important for Byzantine independence. Message validity protects the client's transaction from becoming dependent on a version fabricated by Byzantine replicas; and, by choosing the valid reply with the highest-timestamp, the client is certain to never read a version staler than what it could have read by accessing a single correct replica.

The client then adds the selected (*key, version*) to $ReadSet_T$. If *version* was prepared but not committed, it adds a new write-read dependency to the dependency set Dep_T. Specifically, the client adds to Dep_T a tuple (*version*, $id_{T'}$), which will be used during T's Prepare phase to assert that T is claiming a legitimate dependency, and that the read remains valid. T cannot commit unless all the transactions in Dep_T commit first.

After T has completed execution, the application tells the client whether it should abort T or instead try to commit it:

Abort() The client asks replicas to remove its read timestamps from all keys in $ReadSet_T$. No actions need to be taken for writes, as Basil buffers writes during execution.

Commit() The client initiates the Prepare phase, discussed next, which performs the first phase of the multi-shard two-phase commit (2PC) protocol that Basil uses to commit T.

3.3.2 Prepare Phase

To preserve independent operability, Basil delegates the responsibility for coordinating the 2PC protocol to clients. For a given transaction T, the protocols begins with a Prepare phase, which consists of two stages (Figure 3.1).

In stage ST1, the client collects *commit or abort votes* from each shard that T accesses. Determining the vote of a shard in turn requires collecting votes from all the shard's replicas. To avoid the overhead of coordinating replicas within a shard, Basil lets each replica determine its vote independently, by running a local *concurrency control check*. The flip side of this design is that, since transactions may reach replicas in different orders, even correct replicas within the same shard may not necessarily reach the same conclusion about T. A client C thus tallies replica votes to learn the vote of each shard and, based on how shards voted, decides whether T will commit or abort.

Stage ST2 ensures that C's decision is made durable (or *logged*) across failures. C *logs* the evidence on only one shard. In the absence of contention or failures, Basil's *fast path* guarantees that T's decision is already durable, and this explicit logging step can be omitted, allowing clients to return a commit or abort decision in just a single round trip.

Stage 1: Aggregating votes

1: C → R: Client sends an authenticated ST1 request to all replicas in S.

The message format is ST1 := $\langle$PREPARE, $T\rangle$, where T consists of the transaction's *metadata* := ts_T, $ReadSet_T$, $WriteSet_T$, Dep_T, and of its identifier id_T. To ensure Byzantine clients neither spoof the list of involved shards nor equivocate T's contents, id_T is a cryptographic hash of T's *metadata*.

2: R ← C: Replica R receives a ST1 request and executes the concurrency control check.

Traditional, non-replicated, MVTSO does not require any additional validation at commit time, as transactions are guaranteed to observe *all* the writes that precede them in the serialization order (any "late" write is detected by read timestamps and the corresponding transaction is aborted). This is no longer true in a replicated system: reads could have failed to observe a write performed on a different replica. Basil thus runs an additional concurrency control check to determine whether a transaction T should commit and preserve serializability (Algorithm 1). It consists of seven steps:

Algorithm 1 MVTSO-Check(T)

```
1:  if $ts_T > localClock + \delta$
2:      return Vote-Abort
3:  if $\exists$ invalid $d \in Dep_T$
4:      return Vote-Abort
5:  for $\forall key, version \in ReadSet_T$
6:      if $version > ts_T$  return MisbehaviorProof
7:      if $\exists T' \in Committed \cup Prepared : key \in WriteSet_{T'}$
          $\land\ version < ts_{T'} < ts_T$
8:          return Vote-Abort, optional: ($T'$, $T'$.C-CERT)
9:  for $\forall key \in WriteSet_T$
10:     if $\exists T' \in Committed \cup Prepared$ :
          $ReadSet_{T'}[key].version < ts_T < ts_{T'}$
11:         return Vote-Abort, optional: ($T'$, $T'$.C-CERT)
12:     if $\exists RTS \in key.RTS : RTS > ts_T$
13:         return Vote-Abort
14: $Prepared.add(T)$
15: wait for all pending dependencies
16: if $\exists\ d \in Dep_T : d.decision = Abort$
17:     $Prepared.remove(T)$
18:     return Vote-Abort
19: return Vote-Commit
```

① T's timestamp is within the R's time bound (Lines 1-2).

② T's dependencies are valid: R has either prepared or committed every transaction identified by T's dependencies, and the versions that caused the dependencies were produced by said transactions (Lines 3-4). This ensures that Byzantine clients cannot claim fabricated dependencies that cannot be recovered by the fallback protocol.[2] To limit the impact of cascading aborts, a replica may opt to also reject a transaction whose *dependency depth* is too high, *i.e.*, dependencies that are waiting on dependencies of their own (and so forth).

③ Reads in T did not miss any writes. Specifically, the algorithm (Lines 7-8) checks that there does not exist a write from a committed or prepared transaction T' that (*i*) is more recent than the version that T's read and (*ii*) has a timestamp smaller than ts_T (implying that T should have observed it).

④ Writes in T do not cause reads in other *prepared or committed* transactions to miss a write (Lines 9-11).

⑤ Writes in T do not cause reads in *ongoing* transactions to miss a write: T is aborted if there exists an RTS greater than ts_T (Lines 12-13). This check is optional, and not required for safety; it increases the commit chances of reads.

⑥ T is prepared and made visible to future reads (Line 14).

⑦ All transactions responsible for T's dependencies have reached a decision. R votes to commit T only if all of its dependencies commit; otherwise it votes to abort (Lines 15-19). Aborts caused by cascading dependencies are implicitly durable: R unprepares T and resolves its respective dependents (transactions that depend on T).

3: R → C: Replica returns its vote in a ST1R message.

After executing the concurrency control check, each replica returns to C a Stage 1 reply ST1R:= $\langle id_T, vote \rangle_{\sigma_R}$. A correct replica executes this check **at most once** per transaction and stores its vote to answer future duplicate requests (§ 3.4).

[2]An alternative option is for clients to include the quorum of prepared writes as proof of the dependencies' legitimacy. This allows replicas to accept dependencies that have not been locally observed, at the cost of additional message overhead and signature verification.

4: C ← R: The client tallies replica votes.

C waits for ST1R messages from the replicas of each shard S touched by T. Based on these replies, C determines (*i*) whether S voted to commit or abort; and (*ii*) whether the received ST1R messages constitute a *vote certificate* (V-CERT $:= \langle id_T, S, vote, \{\text{ST1R}\}\rangle$) that proves S's vote to be *durable*. A shard's vote is durable if its original outcome can be independently retrieved and verified at any time by any correct client, independent of Byzantine failures or attempts at equivocation. If so, we dub shard S *fast*; otherwise, we call it *slow*. Votes from a slow shard do not amount to a vote certificate, but simply to a *vote tally*. Though vote tallies have the same structure as a V-CERT, the information they contain is insufficient to make S's vote durable. An additional stage (ST2) is necessary to explicitly make S's vote persistent.

Specifically, C proceeds as follows, depending on the set of ST1R messages it receives (Table 3.1 summarizes the results):

Vote Tally	**Decision**	**Durable**
$3f + 1 \leq$ Commit votes $< 5f + 1$	Commit	✗
$f + 1 \leq$ Abort votes $< 3f + 1$	Abort	✗
$5f + 1$ Commit votes	Commit	✓
$3f + 1 \leq$ Abort votes	Abort	✓
C-CERT for a conflicting transaction	Abort	✓

Table 3.1: Summary of vote tally outcomes.

(1) Commit Slow Path ($3f + 1 \leq$ Commit votes $< 5f + 1$): The client has received at least a *CommitQuorum* (*CQ*) of votes, where $|CQ| = \frac{n+f+1}{2} = 3f + 1$, in favor of committing T. Intuitively, the size of CQ guarantees that two conflicting transactions cannot both commit, since the correct replica that is guaranteed to exist in the overlap of their CQs will enforce isolation. However, C receiving a CQ of Commit votes is not enough to guarantee that another client C', verifying S's vote, would see the same number of Commit votes: after all, f of the replicas in the CQ could be Byzantine, and provide a different vote if later queried by C'. C thus adds S to the set of slow shards, and records the votes it received in the following *vote tally*: $\langle id_T, S, Commit, \{\text{ST1R}\}\rangle$ where $\{\text{ST1R}\}$ is the set of matching (Commit) ST1R replies.

(2) Abort Slow Path ($f + 1 \leq$ Abort votes $< 3f + 1$): A collection of $f + 1$ Abort votes constitutes the minimum *AbortQuorum (AQ)*, *i.e.*, the minimal evidence sufficient for the client to count S's vote as Abort in the absence of a conflicting C-CERT. Requiring an *AbortQuorum* of at least $f + 1$ preserves Byzantine independence: Byzantine replicas alone cannot cause a transaction to abort, as at least one correct replica must have found T to be conflicting with a prepared transaction. However, such *AQ's* are not durable; a client other than C might observe fewer than f abort votes and receive a *CQ* instead. C therefore records the votes collected from S in the following *vote tally*: $\langle id_T, S, Abort, \{\text{ST1R}\}\rangle$ and adds S to the slow set for T.

(3) Commit Fast Path ($5f + 1$ Commit votes): No replica reports a conflict. Furthermore, a unanimous vote ensures that, since correct replicas never change their vote, any client C' that were to step in for C would be guaranteed to observe at least a CQ of $3f + 1$ Commit votes. C' may miss at most f votes because of asynchrony, and at most f more may come from equivocating Byzantine replicas. C thus records the votes collected from S in the following V-CERT: $\langle id_T, S, Commit, \{\text{ST1R}\}\rangle$ and dubs S fast.

(4) Abort Fast Path ($3f + 1 \leq$ Abort votes): T conflicts with a prepared, but potentially not yet committed transaction. S's Abort vote is already durable: since a shard votes to commit only when at least $3f + 1$ of its replicas are in favor of it, once C observes $3f + 1$ replica votes for Abort from S, it is certain that S will never be able to produce $3f + 1$ Commit votes, since that would require a correct replica to change its ST1R vote or equivocate. C therefore creates a V-CERT $\langle id_T, S, Abort, \{\text{ST1R}\}\rangle$, and adds S to the set of fast shards.

(5) Abort Fast Path (One Abort with a c-cert for a conflicting transaction T'): C validates the integrity of the c-cert and creates the following v-cert for S: $\langle id_T, S, Abort, id_{T'}, \text{c-cert}\rangle$. It indicates that S voted to abort T because T conflicts with T', which, as c-cert proves, is a committed transaction. Since c-cert is durable, C knows that the conflict can never be overlooked and that S's vote cannot change; thus, it adds S to the set of fast shards.

After all shards have cast their vote, C decides whether to commit (if all shards voted to commit) or abort (otherwise). Either way, it must make durable the evidence on which its decision is based. As we discussed above, the votes of fast shards already are; if (*i*) there are no slow shards, or (*ii*) a single fast shard voted abort, then, C can move directly to the Writeback Phase (§ **??**): this is Basil's fast path, which allows C to return a decision for T after a single message round trip. If some shards are in the slow set, however, C needs to take an additional step to make its tentative 2PC decision durable in a second phase (st2).

Notably though, Basil does *not* need each slow shard to log its corresponding vote tally in order to make it durable. Instead, Basil first decides whether to commit or abort T based on the shard votes it has received, and then logs its *decision* to only a *single* shard before proceeding to the Writeback phase.

Stage 2: Making the decision durable

5: C → R: The client replicates its tentative 2PC decision durable.

C makes its decision durable by storing an (authenticated) message st2 $:= \langle id_T, decision, \{\text{ShardVotes}\}, view = 0\rangle$ on *one* of the shards that voted in Stage 1 of the Prepare phase; we henceforth refer to this shard, chosen deterministically depending on id_T, as S_{log}. The set {ShardVotes} includes the vote tallies of all shards to prove the decision's validity. Like many consensus protocols (*e.g.*, [30, 101, 141]), Basil relies on the notion of *view* for recovery: the value of *view* indicates whether this ST2 message was issued by the client that initated T (*view*= 0) or it is part of a fallback protocol. We discuss *view*'s role in detail in Section 3.4.

6: R → C: Replicas in S_{log} receive the ST2 message and return ST2R responses.

Each replica validates that C's 2PC decision is justified by the corresponding vote tallies; if so, the replica logs the decision and acknowledges its success. Specifically, it replies to C with a message of the form ST2R := $\langle id_T, decision, view_{decision}, view_{current}\rangle_{\sigma_R}$; $view_{decision}$ and $view_{current}$ capture additional replica state used during recovery. We once again defer an in-depth discussion of views to Section 3.4.

7: C ← R: The client receives a sufficient number of matching replies to confirm a decision was logged.

C waits for $n - f$ ST2R messages whose *decision* and $view_{decision}$ match, and creates a single shard certificate V-CERT$_{S_{log}}$:= $\langle id_T, S, decision, \{$ST2R$\}\rangle$ for the logging shard.

3.3.3 Writeback Phase

C notifies its local application of whether T will commit or abort, and asynchronously broadcasts to all shards that participated in the Prepare phase a corresponding decision certificate (C-CERT for commit; A-CERT for abort).

1: C → R: The client asynchronously forwards decision certificates to all participating shards.

C sends to all involved shards a decision certificate (C-CERT: $\langle id_T$, *Commit*, {V-CERT$_S$}$\rangle$ for a Commit decision, A-CERT: $\langle id_T$, *Abort*, {V-CERT$_S$}$\rangle$ otherwise). We distinguish between the fast, and slow path: On the fast path, C-CERT consists of the full set of Commit

V-CERT votes from all involved shards, while an A-CERT need only contain one V-CERT vote for Abort. On the slow path, both C-CERT and A-CERT simply include V-CERT$_{S_{log}}$.

2: R ← C: Replica validates C-CERT or A-CERT and updates store accordingly.

Replicas update all local data structures, including applying writes to the datastore on commit and notifying pending dependencies.

3.3.4 MTSO Execution Example

Figure 3.2 illustrates an example execution in Basil and the respective MVTSO-check results across several inconsistent replicas. Replicas manage two objects, *a* and *b*, whose initial versions vary across replicas. The example follows the life of four transactions; for simplicity, we identify a transaction via its timestamp.

① Transaction T_{10} ($ts_T = 10$) reads two different committed versions (a_6 and a_8) and selects the freshest version (a_8).

② T_{10} and T_9 are conflicting: T_{10} missed T_9's write (a_9), and as a result, one of the transactions has to abort. This decision can differ across replicas, as they may process T_9 and T_{10} in different orders. Replicas prepare only the first transaction processed and vote to abort the other.

③ T_7 too writes to *a* but is not in conflict with T_{10}, since T_{10}'s read version a_8 is fresher than T_7's write a_7.

④ T_{10} prepares successfully at several replicas; T_{12} reads the prepared write b_{10} and acquires a dependency on T_{10}.

⑤ T_{10} successfully commits, but its commit confirmation (Writeback) has not yet arrived at all replicas: ⑥ T_{12} cannot prepare at the first replica (validation blocks) until its dependency (T_{10}) is resolved locally.

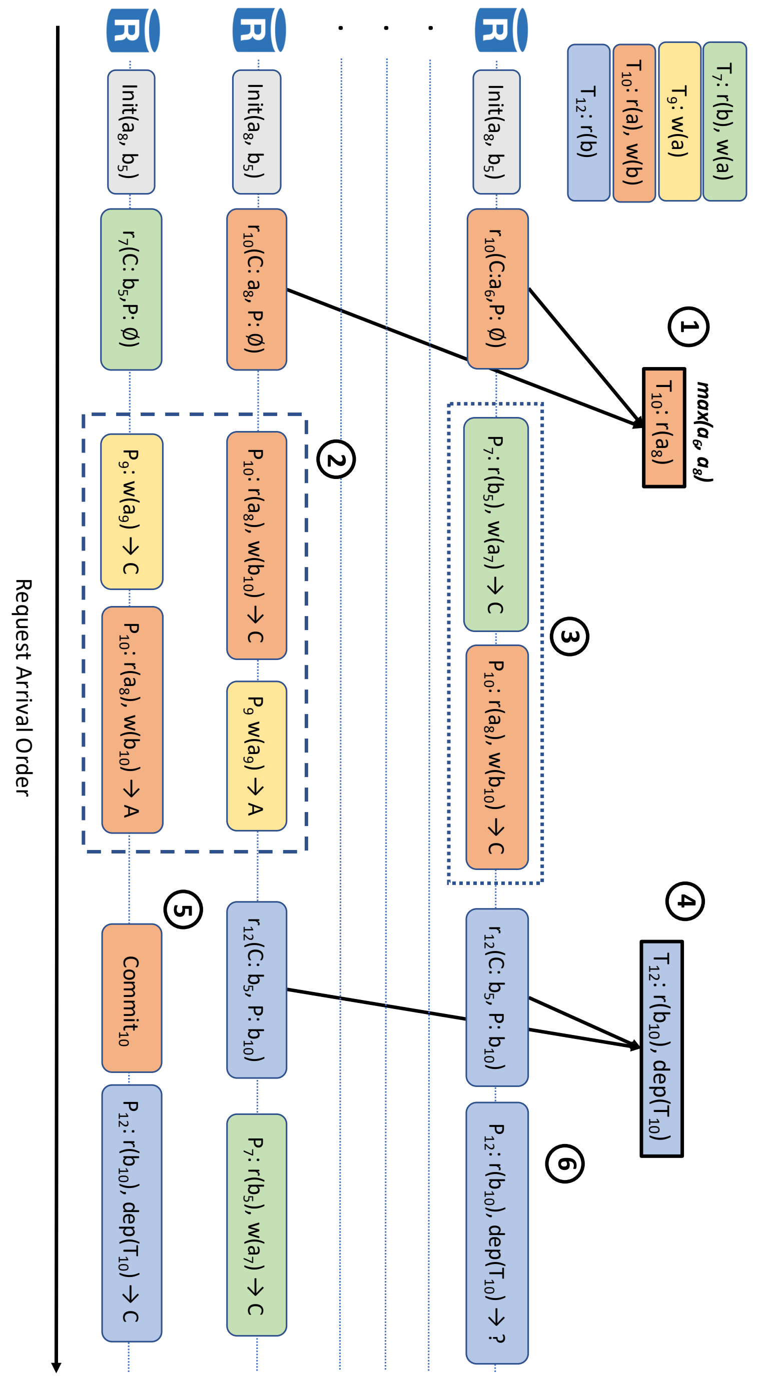

Figure 3.2: Basil example execution. $r_x(C : a_y, P : a_z)$ denotes that transaction T_x—with timestamp x—reads the version y of object a written by committed transaction T_y and version z from tentatively prepared transaction T_z. $P_x(r(a_z), w(b_x), dep(T_z))$ denotes a transaction T_x's prepare request with ReadSet a_z, the WriteSet b_x, and a dependency on T_z. $\rightarrow C/A$ denotes the prepare request's validation outcome (commit/abort).

3.3.5 An Optimization: Reply Batching

To amortize the cost of signature generation and verification, Basil batches messages (Figure 3.3). Unlike leader-based systems, Basil has no central sequencer through which to batch requests; instead, it implements batching at the replica *after* processing messages. To amortize signature generation for replies, Basil replicas create batches of b request replies, generate a Merkle tree [128] for each batch, and sign the root hash. They then send to each client C that issued a request: (i) the root hash *root*, (ii) a signed version σ of the same *root*, (iii) the appropriate request reply R_C, and (iv) all intermediate nodes (denoted π_C in Figure 3.3) necessary to reconstruct, given R_C, the root hash *root*. Through this batching, the cost of signature generation is reduced by a factor of b, at the cost of $log(b)$ additional messages.

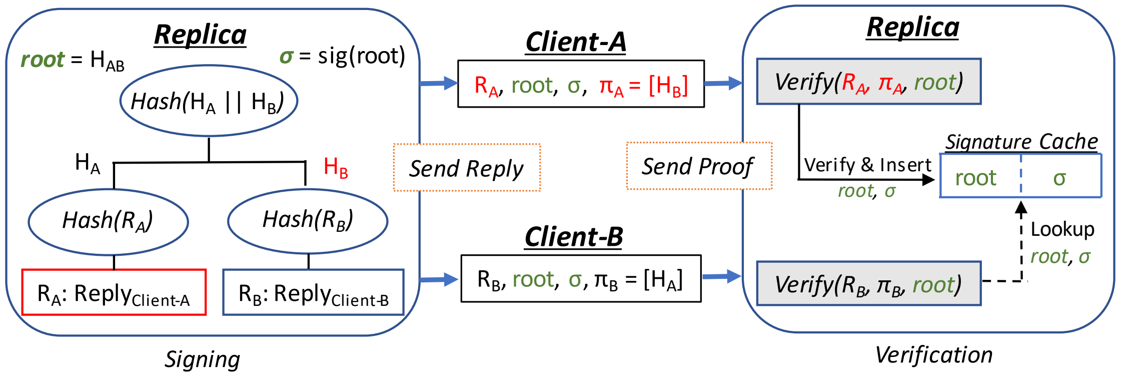

Figure 3.3: Basil batching for two clients. Signature σ and batch root are the same across batched replies. Reply R_C and proof π_C are unique to each client C and can validate root.

To amortize signature verification, Basil uses caching. When a replica successfully verifies the root hash signature in a client message m, it caches a map between the corresponding root hash value and the signature. If the replica later receives a message m' carrying the same root hash and signature as m (indicating that m and m' refer to the same batch of replies), it can, upon verifying the correctness of the root hash, immediately declare the corresponding signature valid.

3.3.6 Discussion

Stripping layers When 2PC is layered above shards that already order transactions internally using state machine replication, then *within every shard* every correct replica has logged the vote of every other correct replica. Basil's design avoids this indiscriminate cost: if all shards are fast, then their votes are already durable without requiring replicas to run any coordination protocol; and if some shards are slow, as we mentioned above, only the replicas of a single shard need to durably log the decision. As a result, the overhead of Basil's logging phase (Stage 2) remains constant in the number of involved shards.

Signature Aggregation Basil, like recent related work [80, 192], can make use of signature aggregation schemes [24–26, 29, 69, 90, 129, 163] to reduce total communication complexity. The client could aggregate the (matching) signed ST1R or ST2R replies into a single signature, thus ensuring that all messages sent by the client remain constant-sized, and hence Basil total communication complexity can be made linear. In practice, however, most signature aggregations schemes are only efficient for large f [80]. The current Basil prototype does not implement this optimization.

Not all signature aggregation schemes are compatible with batching. *Multi-signatures* [24, 25, 90, 129] and *Threshold signatures* [24, 29, 163], for instance, cannot be used together with batching as they require the signed messages to match. Batch roots, instead, can differ across replicas who may not create common batches. Aggregating signed batch roots requires schemes that support aggregating signatures of distinct messages [26, 69].

Why $n = 5f + 1$ replicas per shard? Using fewer replicas has two main drawbacks. First, it eliminates the possibility of a commit fast path. With a smaller replication factor, *CQ*s of size $n-2f$ (f can differ because of asynchrony, another f can differ because of equivocation) would no longer be guaranteed to overlap in a correct replica, making it possible for conflicting transactions to commit, in violation of Byz-serializability. Second, it precludes Byzantine independence. For progress, clients must always be able to observe either a *CQ* or an *AQ*, but, for Byzantine independence, the size of neither quorum must fall below $f+1$: with $n \leq 5f$, it becomes impossible to simultaneously satisfy both requirements. We discuss in Section 3.7 the conceptual modifications required to instantiate Basil with only $3f + 1$ replicas.

3.4 Transaction Recovery

For performance, Basil optimistically allows transactions to acquire dependencies on uncommitted operations. Without care, Byzantine clients could leverage this optimism to cause transactions issued by correct clients to stall indefinitely. To preserve Byzantine independence, transactions must be able to eventually commit even if they conflict with, or acquire dependencies on, stalled Byzantine transactions. To this effect, Basil enforces the following invariant: if a transaction acquires a dependency on some other transaction T, or is aborted because of a conflict with T, then a correct participant (client or replica) has enough information to successfully complete T.

Specifically, Basil clients whose transactions are blocked or aborted by a stalled transaction T try to finish T by triggering a *fallback* protocol. To this end, Basil modifies MVTSO to make visible the operations of transactions that have *prepared* only. As T's ST1 messages contain all of T's planned writes, any client or replica can use this information to take it upon itself to finish T. A correct client is guaranteed to be able to retrieve the ST1 for any of its dependencies, since $f+1$ replicas (*i.e.*, at least one correct) must have implicitly vouched for that ST1 during T's read phase. Likewise, a correct client's transaction only aborts if at least $f + 1$ replicas report a conflict.

Basil's fallback protocol starts with clients: any client blocked by a stalled transaction T can try to finish it. In the **common case**, it will succeed by simply re-executing the previously described Prepare phase; success is guaranteed as long as replicas within the shard S_{log} that logged shard votes in Stage 2 of T's Prepare phase store the same decision for T.

The **divergent case**, in which they do not, can occur in one of two ways: (*i*) a Byzantine client issued T and sent deliberately conflicting ST2 messages to S_{log}; or (*ii*) multiple correct clients tried to finish T concurrently, and collected Prepare phase votes (set of ST1R messages) that led them to reach (and try to store at S_{log}) different decisions. Fortunately, in Basil a Byzantine client cannot generate conflicting ST2 messages at will: its ability to do so depends on the odds of receiving, from the replicas of at least one shard, votes that constitute *both* a CQ and an AQ (*i.e.*, *3f+1* Commit votes and *f+1* Abort votes)—odds which our evaluation (§ 3.6) suggests are low.
Whatever the cause, if a client trying to finish T observes that replicas in S_{log} store different decisions, it proceeds to elect a *fallback leader*, chosen deterministically among the replicas in S_{log}. Through this process, Basil guarantees that clients are always able to finish dependent transactions after at most $f + 1$ leader elections (since one of them must elect a correct leader).

Though Basil's fallback protocol is reminiscent of the traditional view-change protocols used to evict faulty leaders, it differs in three significant ways. First, it requires no leader in the common case; further, if electing a fallback leader becomes necessary, communication costs can be made linear in the number of replicas using signature aggregation schemes (§ 3.3.6). Second, the fallback election is transaction-local, and affects only transactions that access the same operations as the stalled transaction: when a fallback leader is elected for T, the scope of its leadership is limited to finishing T. In contrast, a standard view-change prevents the system from processing *any* operation and the leader, once elected, lords over all consensus operations during its tenure. Finally, Basil's fallback leaders have no say on the ordering of transactions or on the decision they recover; consequently, they do not compromise Byzantine independence.

As in traditional view-change protocols, each leader operates in a *view*. For independent operability, views are defined on a per-transaction basis. Transactions start in *view* = 0; transactions in that view can be brought to a decision by any client. A replica increases its view number for T each time it votes to elect a new fallback leader.

We now describe the steps of the fallback protocol triggered by a client C wishing to finish a transaction T, distinguishing between the aforementioned common and divergent cases.

Common case In the common case, the client simply resends a ST1 message (renamed for clarity *Recovery Prepare* (RP) in this context) to all the replicas in shards accessed by T. Replicas reply with an RPR message which, depending on the progress of previous attempts (if any) at completing T corresponds to either (*i*) a ST1R message; (*ii*) a ST2R message; or (*iii*) a C-CERT or A-CERT certificate. Based on these replies, the client can fast-forward to the corresponding next step in the Prepare or Writeback protocol. In the common case, stalled dependencies thus cause correct clients to experience only a single additional round-trip on the fast path, and at most two if logging the decision is necessary (slow path).

Divergent case If, however, the client only receives non-matching ST2R replies, more complex remedial steps are needed. ST2R can differ (*i*) in their decision value and (*ii*) in their view number $view_{decision}$. The former, as we saw, is the result of either explicit Byzantine equivocation or of multiple clients attempting to concurrently terminate T. The latter indicates the existence of prior fallback invocations: a Byzantine fallback leader, for instance, could have intentionally left the fallback process hanging. In both scenarios, the client elects a new fallback leader. The steps outlined below ensure that, once a correct fallback leader is elected, replicas can be reconciled without introducing live-lock.

(1: C → R): Upon receiving non-matching ST2R responses, a client starts the fallback process.

The client sends INVOKEFB := $\langle id_T,\ views \rangle$, where *views* is the set of signed current views associated with the RPR responses received by the client.

(2: R → R$_{\mathbf{FL}}$): Replicas receive fallback invocation INVOKEFB and start election of a fallback leader R$_{\mathbf{FL}}$ for the current view.

R takes two steps. First, it determines the most up-to-date view held by correct replicas in S_{log} and adopts it as its current view $view_{current}$. Second, R sends message ELECTFB := $\langle id_T, decision, view_{current} \rangle_{\sigma_R}$ to the replica with id $v_{current} + (id_T \mod n)$ to inform it that R now considers it to be T's fallback leader.

R determines its current view as follows: If a view v appears at least $3f + 1$ times in the current *views* received in INVOKEFB, then R updates its $view_{current}$ to $max(v + 1, view_{current})$; otherwise, it sets its $view_{current}$ to the largest view v greater than its own that appears at least $f + 1$ times in *current views*. When counting how frequently a view is present in the received *current views*, R uses vote subsumption: the presence of view v counts as a vote also for all $v' \leq v$.

The thresholds Basil adopts to update a replica's current view are chosen to ensure that all $4f + 1$ correct replicas in S_{log} quickly catch up (in case they differ) to the same view, and thus agree on the identity of the fallback leader. Specifically, by requiring $3f + 1$ matching views to advance to a new view v, Basil ensures that at least $2f + 1$ correct replicas are at most one view behind v at any given time. In turn, this threshold guarantees that (*i*) a correct client will receive at least $f+1$ matching views for $v' \geq v-1$ in response to its RP message and (*ii*) will include them in its InvokeFb. These $f + 1$ matching views are sufficient for all $4f + 1$ correct replicas to catch up to view v', then (if necessary) jointly move to view v, and send election messages to the fallback leader of v. We defer additional details and proofs to Section 3.5.

(3: $\mathbf{R_{FL}} \rightarrow \mathbf{R}$): Fallback leader R_{FL} aggregates election messages and sends decisions to replicas.

R_{FL} considers itself elected upon receiving $4f + 1$ ElectFb messages with matching views $view_{elect}$. It proposes a new decision $dec_{new} = majority(\{decision\})$ and broadcasts message DecFb $:= \langle (id_T, dec_{new}, view_{elect})_{\sigma_{R_{FL}}}, \{\text{ElectFb}\}\rangle$, which includes the ElectFb messages as proof of its leadership.

Importantly, an elected leader can only propose *safe* decisions: if a decision has previously been returned to the application or completed the Writeback phase, it must have been *logged* successfully,*i.e.*, signed by at least $n - f = 4f + 1$ replicas. Thus, in any set of $4f+1$ ElectFb messages the decision must appear at least $2f+1$ times, *i.e.*, a majority. Note that this condition no longer holds when using fewer that $5f + 1$ replicas per shard (§ 3.7): using a smaller replication factor would require (*i*) an additional (third) round of communication, and (*ii*) including proof of this communication in all replica votes (an $O(n)$ increase in complexity), to guarantee that conflicting decision values may not be logged for the same transaction.

(4: R → C): Replicas sends a st2r message to interested clients.

Replicas receive a DecFb message and adopt the message's decision (and $view_{decision}$)

as their own if their current view is smaller or equal to $view_{elect}$ and the proof is valid. If so, replicas update their current view to $view_{elect}$ and forward the decision to all interested clients in a ST2R message: $\langle id_T, decision, view_{decision}, view_{current}\rangle_{\sigma_R}$.

(5: C): A client creates a V-CERT or restarts fallback.

If the client receives $n - f$ ST2R with matching decision and decision views, she creates a V-CERT$_{S_{log}}$ and proceeds to the Commit phase. Otherwise, it restarts the fallback using the newly received $view_{current}$ messages to propose a new view.

An Example We illustrate the entire divergent case algorithm in Figure 3.4, which for simplicity considers a transaction T involving a single shard. With multiple shards, only the common case RP messages (and replies) would involve all shards; the divergent case would always touch only a single shard, S_{log}.

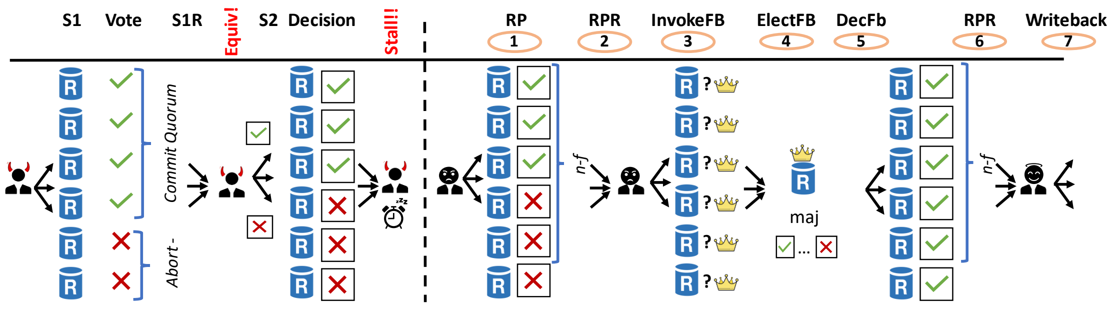

Figure 3.4: Fallback protocol overview. A Byzantine client equivocates ST2R *decisions and stalls. An interested client invokes the fallback protocol: the elected fallback leader reconciles* ST2R *decisions, allowing the interested client to commit the stalled transaction.*

To begin the process of committing T, a (Byzantine) client broadcasts message ST1 and waits for all ST1R messages. Since the replies it receives allow it to generate both a Commit and Abort quorum, the client chooses to equivocate, sending ST2 messages for both Commit and Abort. It then stalls. A second correct client who acquired a dependency on T attempts to finish it. It sends RP messages (①) and receives non-matching RPR messages (three Commit and two Abort decisions, all from view 0) (②).

To redress this inconsistency, the correct client invokes a Fallback with view 0 (③). Upon receiving this message, replicas transition to view 1 and send their own decision to view 1's leader in an ElectFb message (④). In our example, having received a majority of Commit decisions, the leader chooses to commit and broadcasts its decision to all other replicas in a DecFb message (⑤). Finally, replicas send the transaction's outcome to the interested client (⑥), who then proceeds to the Writeback phase (⑦).

3.5 Correctness

We first sketch the main theorems and lemmas necessary to show safety and liveness for Basil; full proofs follow. We proceed in two steps: we first prove safety without the fallback protocol. We then extend our correctness argument to handle fallback cases.

First, we prove that each replica generates a locally serializable schedule. Specifically, we show that at every correct replica, the set of transactions for which the MVTSO-Check returns Vote-Commit forms an acyclic serialization graph [4].

Lemma 1 *On each correct replica, the set of transactions for which the MVTSO-Check returns Vote-Commit forms an acyclic serialization graph.*

We then show that decisions for transactions are unique.

Lemma 2 *There cannot exist both an* C-CERT *and a* A-CERT *for a given transaction.*

Next, we show that Byz-serializability is preserved *across* replicas. In particular, we show that two pairwise conflicting transactions cannot both commit.

Lemma 3 *If T_i has issued a* C-CERT *and T_j conflicts with T_i, then T_j cannot issue a* C-CERT.

Given these three lemmas, we prove that Basil satisfies Byz-serializability.

Theorem 1 *Basil maintains Byz-serializability.*

Finally, we show that Basil preserves Byzantine independence under non-adversarial network assumptions.

Theorem 2 *Basil maintains Byzantine independence in the absence of a network adversary.*

We now explicitly consider the fallback protocol. We first show that certified decisions remain durable.

Lemma 4 *Fallback leaders cannot propose decisions that contradict an existing* C-CERT/A-CERT.

Additionally, we show that any reconciled decision must have been proposed by a client.

Lemma 5 *Any decision proposed by a fallback leader was proposed by a client.*

Given these two Lemmas we conclude:

Theorem 3 *Invoking the fallback mechanism preserves Theorem 1 and Theorem 2.*

Next we show that, given partial synchrony [54] and the existence of a global stabilization time (GST), Basil's fallback mechanism guarantees progress for correct clients.

Theorem 4 A correct client can reconcile correct replicas' views in at most two round-trips and one timeout.

We use this result to prove that fallback election reliably succeeds.

Lemma 7 *After GST, and in the presence of correct interested clients, a correct fallback leader is eventually elected.*

Given this lemma, we show that Basil allows correct clients to complete their dependencies during synchronous periods.

Theorem 5 *A correct client eventually succeeds in acquiring either a* C-CERT *or* A-CERT *for any transaction of interest.*

3.5.1 Byz-Serializability

We show that the set of committed transactions is Byz-serializable. By design, the transactions of correct client's only read from committed writes (*i.e.*, they are legal); transactions that read prepared data wait for dependencies to commit before commiting themselves (Alg. 1, Lines 15-19). We now show that the set of all committed transactions forms an acyclic serialization graph, and thus is equivalent to some serial execution [4].

Lemma 1 *On each correct replica, the set of transactions for which the MVTSO-Check returns Vote-Commit forms an acyclic serialization graph.*

Proof. We use Adya's formalism here [4]. An execution of Basil produces a direct serialization graph (DSG) whose vertices are committed transactions, denoted T_t, where t is the unique timestamp identifier. Edges in the DSG are one of three types:

- $T_i \xrightarrow{ww} T_j$ if T_i writes the version of object x that precedes T_j in the version order.
- $T_i \xrightarrow{wr} T_j$ if T_i writes the version of object x that T_j reads.
- $T_i \xrightarrow{rw} T_j$ if T_i reads the version of object x that precedes T_j's write.

We assume, as does Adya, that if an edge exists between T_i and T_j, then $T_i \neq T_j$.

First, we prove that if there exists an edge $T_i \xrightarrow{rw/wr/ww} T_j$, then $i < j$. We consider each case individually.

$\xrightarrow{ww}$ **case.** Assume that there is a $T_i \xrightarrow{ww} T_j$ edge. This means that T_i writes a version of object x that precedes T_j in the version order. MVTSO's version order, for each object, is equivalent to the timestamp order of the transactions that write to the object. Note that the order in which versions are added to the list may not match the timestamp order, but the versions are added to the appropriate indices in the list such that the timestamps of all preceding versions are smaller and the timestamps of all subsequent versions are larger than the new version's timestamp. This implies that $i < j$.

$\xrightarrow{wr}$ **case.** Assume that there is a $T_i \xrightarrow{wr} T_j$ edge. This means that T_j reads a version of object x written by T_i. MVTSO's algorithm returns, for a read in T_j, the latest version of an object whose version timestamp is lower than j. This implies that $i < j$.

$\xrightarrow{rw}$ **case.** This case is the most complex. We prove it by contradiction. Assume that there is a $T_i \xrightarrow{rw} T_j$ edge such that this edge is the first edge where $i \geq j$. $T_i \xrightarrow{rw} T_j$ means that T_i reads the version of object x that precedes T_j's write. Let that version be x_k. By the assumption that $T_i \neq T_j$, $i > j$. By the definition of the direct serialization graph, $T_k \xrightarrow{wr} T_i$ and $T_k \xrightarrow{ww} T_j$. The previous cases imply that $k < i$ and $k < j$.

Consider a replica that ran the MVTSO-Check for both T_i and T_j. There are two subcases: either the check for T_i was executed before the check for T_j or vice versa.

1. T_i **before** T_j**.** T_i must have committed because it exists in the DSG. This implies that T_i was in the *Prepared* set when the check was executed for T_j (Line 14 of Algorithm 1). When the check for T_j reached Line 10 of Algorithm 1 for T_j's

write of x_j, the condition was satisfied by T_i's read of x_k because $k < j$ and $j < i$. Therefore, the check for T_j returned Vote-Abort. However, this is a contradiction because T_j committed.

2. T_j **before** T_i**.** T_j must have committed because it exists in the DSG. This implies that T_j was in the *Prepared* set when the check was executed for T_i. When the check for T_i reached Line 7 of Algorithm 1 for T_i's read of x_k, the condition was satisfied by T_j's write of x_j because $k < j$ and $j < i$. Therefore, the check for T_i returned Vote-Abort. However, this is a contradiction because T_i is committed.

In all cases, the preliminary assumption leads to a contradiction. This implies that $i < j$.

Next, we use the fact that if there exists an edge $T_i \xrightarrow{rw/wr/ww} T_j$, then $i < j$ to prove that the set of transactions for which MVTSO-Check returns Vote-Commit is serializable.

Acyclicity The set of transactions is serializable if the DSG has no cycles. Assume for a contradiction that there exists a cycle consisting of n transactions $T_{ts_1}, ..., T_{ts_n}$. By the previous fact, this implies that $ts_1 < ... < ts_n < ts_1$. However, transaction timestamps are totally ordered. This is a contradiction. Thus, the DSG has no cycles. □

Lemma 2 *There cannot exist both an* C-CERT *and a* A-CERT *for a given transaction.*

Proof. There are two ways that the client can form a C-CERT/A-CERT: through the single phase fast-path, and through the two-phase slow path. We consider both in turn. Recall that on the fast path a C-CERT contains a list of ST1R as evidence for each shard, while an A-CERT contains a list of ST1R for a single shard (that voted to abort). On the slow path instead, both a C-CERT and an A-CERT contain only a list of ST2R only for the logging shard S_{log}. The intuition behind this difference is that, on the fast path, the two-phase commit decision has yet to be validated, whereas on the slow path, it has already been confirmed and logged to a single shard in an effort to reduce redundant logging.

In the following, we show that for all possible combinations of certificates, no C-CERT and A-CERT can co-exist.

Commit Fast Path, Abort Fast Path Assume that a client generates both a C-CERT and an A-CERT for T and that both went fast path. A fast C-CERT requires $n = 5f + 1$ replicas (commit fast path quorum) to vote commit T on *every* shard, while a fast A-CERT requires either 1 vote if a C-CERT is present to prove the conflict (abort fast path quorum, case 1), or $3f + 1$ votes on a *single* shard otherwise (abort fast path quorum, case 2). We distinguish abort fast path quorum cases 1 and 2:

1. If a C-CERT exists for a conflicting transaction, then, by definition of a C-CERT, at least $3f + 1$ (commit slow path quorum) replicas of the shard in question must have voted to commit a transaction T' that conflicts with T. Since correct replicas (*i*) do not vote to commit conflicting transactions (Lemma 1), and (*ii*) never change their vote, it follows that at least $2f + 1$ correct replicas must vote to abort T. By assumption, however, all correct replicas voted to commit T. We have a contradiction.
2. Correct replicas never change their vote. Because any two replica quorums (on a given shard) of size $5f + 1$ and $3f + 1$, respectively, must intersect in at least correct replica, it follows that at least one correct replica equivocated and voted to both commit and abort. We have a contradiction.

Abort Fast Path, Commit Slow Path Assume that a client generates both a C-CERT that went slow path and an A-CERT that went fast path for T. A slow C-CERT requires $n - f = 4f + 1$ matching ST2R replies from the logging shard. In order for a correct replica to send a commit ST2R message it must have received a vote tally of at least $3f + 1$ commit votes created on *every* shard. Instead, a fast A-CERT requires either 1 vote if a C-CERT is present to prove the conflict (abort fast path quorum, case 1), or $3f + 1$ votes on a *single* shard otherwise (abort fast path quorum, case 2). We distinguish abort fast path quorum cases 1 and 2:

1. If a C-CERT exist for a conflicting transaction, then by definition of a C-CERT, at least $3f + 1$ (commit slow path quorum) replicas of the shard in question must have voted to commit a transaction T' that conflicts with T. Since correct replicas (*i*) do not vote to commit conflicting transactions (Lemma 1), and (*ii*) never change their vote, it follows that at least $2f + 1$ correct replicas must vote to abort T. By assumption, however, at least $3f + 1$ replicas voted to commit T. Because any two replica quorums (on a given shard) consisting of $2f + 1$ correct and $3f + 1$ replicas, respectively, must intersect in at least correct replica, it follows that at least one correct replica equivocated and voted to both commit and abort. We have a contradiction.
2. Correct replicas never change their vote. Because any two sets of $3f + 1$ replicas (on a given shard) must intersect in at least correct replica, it follows that at least one correct replica equivocated and voted to both commit and abort. We have a contradiction.contradiction.

Commit Fast-Path, Abort Slow-Path Assume that a client generates a C-CERT that went fast path and a A-CERT that went slow path for a transaction T. A fast C-CERT requires $n = 5f + 1$ replicas (commit fast path quorum) to vote to commit T on *every shard*, while a slow A-CERT requires $n - f = 4f + 1$ ST2R messages from the logging shard with the decision to abort. In order for a correct replica to send a valid abort ST2R message, it must have received a vote tally of at least $f + 1$ abort votes created on a single shard.

Correct replicas never change their vote. Because any two replica quorums (on a given shard) of size $5f + 1$ and $f + 1$, respectively, must intersect in at least correct replica, it follows that at least one correct replica equivocated and voted to both commit and abort. We have a contradiction.

Commit Slow-Path, Abort Slow-Path Assume that a client generates a C-CERT and an A-CERT for a transaction T, both of which went slow path. Recall that each slow path certificate requires $n-f = 4f+1$ matching ST2R replies from the logging shard. Because correct replicas adopt at most one decision per view and any two replica quorums of size $4f + 1$ must overlap in at least one correct replica it follows that there cannot exist a C-CERT and A-CERT for the same view. Without loss of generality, assume that the C-CERT was generated in a view smaller than the A-CERT. By Lemma 4, however, the fallback protocol cannot recover a decision that conflicts with an existing C-CERT or A-CERT. We have a contradiction.

It follows that there cannot be both a C-CERT and A-CERT for a transaction T. □

It follows from Lemma 2 that no two correct replicas can ever process a different outcome (commit/abort) for a transaction T. Thus, given a fixed set of total transactions, all replicas are eventually consistent.

Next, we show that two conflicting transactions cannot both commit. We define transaction T_j as *conflicting* with T_i if adding T_j to a serialization graph that includes T_i would introduce a cycle. By Lemma 1, if a replica has committed or prepared T_i (*i.e.*, $T_i \in Commit \cup Prepared$), then its local MVTSO-check will vote to abort T_j. We show that Basil also precludes such conflicts *across* replicas.

Lemma 3 *If T_i has issued a* C-CERT *and T_j conflicts with T_i, then T_j cannot issue a* C-CERT.

Proof. As T_i has committed, a client has generated a C-CERT for T_i. If a C-CERT for T exists, then *at every involved shard* at least $3f + 1$ replicas voted to commit the transaction (the slow path requires 3f+1 votes, the fast path 5f+1), and consequently at least $2f + 1$ correct replicas on each involved shard voted to commit.

Assume by way of contradiction that a client C has created a C-CERT for T_j. There are two ways of achieving this: either C generates a C-CERT through the fast path or it does so through the slow path. Note, that since T_i and T_j conflict, their involved shards must intersect in one common shard S_{common}.

Fast Path If T_j's C-CERT was created through the fast path, then all $n - f = 4f + 1$ correct replicas on S_{common} voted to commit T_j. By assumption, however, at least $2f + 1$ correct replicas have already voted to commit T_i and thus vote to abort T_j (Lemma 1). We have a contradiction. C cannot generate a C-CERT for T_j through the fast path.

Slow Path If T_j's C-CERT was created through the slow path, then there must exist at least $2f + 1$ correct replicas on S_{common} that voted to commit T_j. By assumption, however, at least $2f + 1$ correct replicas have already voted to commit T_i. Because any two sets of $2f + 1$ correct replicas must intersect (there are $4f + 1$ total correct replicas) it follows that at least one correct replica voted to commit T_j despite it creating a cycle in the serialization graph with T_i (violating Lemma 1). We have a contradiction. C cannot generate a C-CERT for T_j through the slow path. □

Theorem 1 *Basil maintains Byz-serializability.*

Proof. Consider the set of transactions for which a C-CERT could have been assigned. Consider a transaction T in this set. By Lemma 2, there cannot exist an A-CERT for this transaction. By Lemma 3, there cannot exist a conflicting transaction T' that generated a C-CERT. Consequently, there cannot exist a committed transaction T' in the history. The history thus generates an acyclic serialization graph. The system is thus Byz-Serializable. □

3.5.2 Byzantine Independence

Theorem 2 *Basil maintains Byzantine independence in the absence of network adversary.*

We show that correct client's transaction results (reads and commit decisions) cannot be unilaterally decided by any group of (colluding) Byzantine participants.

Proof. First, we note that a set of Byzantine replicas cannot force a correct client to read (*i*) an imaginary version or (*ii*) a stale version. Specifically, a correct client reads committed writes only if they are associated with a valid C-CERT. Moreover, clients read the freshest among at least $f + 1$th read versions returned. This ensures that the returned version is at least as recent as what could have been written by a single correct replica. Likewise, a client only reads uncommitted (prepared) writes if they were returned by $f + 1$ replicas.

Next, we show that transaction decisions cannot be made by Byzantine replicas alone. Both fast and slow path commit and abort quorums contain at least one correct replica. Consequently, if a transaction commits (or aborts), at least one correct replica voted accordingly.

Finally, we observe that although a Byzantine client may choose to sabotage its own transaction success (*e.g.*, through stale reads or poor timestamp selection), it cannot influence the outcome of its transaction *after* submitting it for validation. In particular, a Byzantine client cannot unilaterally abort its own transactions once it has prepared. Because only prepared (or committed) transactions are visible it follows that a Byzantine client cannot unilatterally cause dependent (correct) transactions to abort. □

We note that these safeguards are not sufficient to guarantee Byzantine independence when an adversary controls the network. A powerful network adversary, for instance, could systematically delay delivery or dynamically inject conflicting transactions to influence the outcomes of target transactions.

3.5.3 Fallback Safety

In the absence of an elected fallback leader (**common case** of transaction recovery), clients use ST1 or ST2 messages and follow the same rules as during normal execution. Byzantine clients that equivocate during ST2 are indistinguishable from concurrently active clients. As such, all theorems introduced so far in Section 3.5 continue to hold.

A fallback leader is only elected when a client detects inconsistent ST2R messages (the **divergent case** of transaction recovery) during decision logging. Consequently, we only need to consider the effects of the fallback protocol on decisions that were generated through the slow path. Note that slow-path processing (slow path C-CERT's and A-CERT's) and the fallback leaders operate only on the logging shard; we thus consider only the logging shard in the rest of our proofs. We show that (*i*) if a slow path C-CERT or A-CERT already exists, the fallback protocol never produces a conflicting decision certificate, and that (*ii*) if no decision certificate exists, any decision C-CERT/A-CERT created must correspond to a decision made by some client (*i.e.*, the decision is based on a set of SHARDVOTES).

Reminder. For convenience, we re-state the decision reconciliation rule used by the fallback. The fallback leader collects ElectFb messages from $n - f = 4f + 1$ replicas, and re-proposes the majority decision: $dec_{new} = maj(\{\text{ElectFb}.decision\})$. We note that matching views are not required here.

We begin by showing:

Lemma 4 *Fallback leaders cannot propose decisions that contradict an existing* c-cert/a-cert.

Proof. By Lemma 2, in the absence of the fallback protocol, a c-cert and a-cert cannot co-exist for any given transaction. In particular, existence of a fast c-cert implies that that commit is the only valid decision for a st2 message (and vice versa abort for a fast a-cert). By design, the fallback protocol operates only on Stage 2 decisions: it requires replicas to have received st2 messages, and only ever changes slow path decisions. For simplicity, we thus consider in the following only slow path c-certs and a-certs.

For any existing slow path c-cert/a-cert, the associated decision must have been adopted and logged by at least $3f + 1$ correct replicas (through st2 and st2r messages) in a matching decision view v. Without loss of generality, let the decision be Commit (the reasoning for Abort is identical) and the corresponding decision certificate be a c-cert. Because $4f + 1$ matching replies are necessary to form the c-cert, it follows that at least $3f + 1$ correct nodes have logged Commit as their decision in view v. Correct replicas never change their vote within a view.

Let R_{FB} be the **first** fallback leader to be elected for a view $v' > v$. At least $3f + 1$ correct replicas must be in v', since $4f + 1$ ElectFb messages are required for a leader to be elected in view v' (property P1). We note that any existing c-cert with view $v \leq v'$ must have been constructed from st2r messages sent by correct replicas **before** moving to view v', since correct replicas do not accept decisions from smaller views. Since R_{FB} is the *first* fallback leader, the $3f + 1$ correct replicas that logged Commit have never changed their decision ((property P2)).

By P1 it holds that at least $3f + 1$ correct replicas must have cast an ElectFb message for v', and by P2 at least $3f + 1$ correct replicas have logged Commit as their latest decision. By quorum intersection, it follows that any ElectFb quorum in view v' ($4f + 1$ ElectFb votes) contains at least $2f + 1$ Commit decisions (any two sets of $3f + 1$ correct replicas overlap in $2f + 1$ correct replicas). It follows from the decision reconciliation rule that the fallback leader R_{FB} in view v' must re-propose Commit (the majority decision).

By induction, this holds for all consecutive views and fallback leaders: if there ever existed $3f + 1$ correct replicas that logged decision $dec_v = d$ in view v, then correct replicas will only ever log $dec_{v'} = d$ for views $v' > v$, since a fallback leader will always observe a majority for decision d.

Consequently, Lemma 4 holds. □

Next, we show:

Lemma 5 *Any decision proposed by a fallback leader was proposed by a client.*

Proof. Let R_{FB} be the **first** elected fallback leader that proposes a decision (let its associated view be v).

By design, a correct replica only sends a message ElectFb once it has logged a decision. Thus, R_{FB} is guaranteed to receive a quorum of $4f + 1$ ElectFb messages all containing decisions. As the fallback waits for $4f + 1$ messages, one decision must be in the majority. By the decision reconciliation rule, R_{FB} proposes the majority decision.

By assumption, R_{FB} is the first fallback leader to propose a decision. Thus, all decisions included in correct replicas' ElectFb messages were (*i*) made by a client, and (*ii*) are consistent with the provided ShardVotes (correct replicas will verify that st2 messages have sufficient evidence for the decision).

Since any majority decision requires at least $2f + 1$ matching ElectFb messages, it follows that at least one was created by a correct client and is hence valid. Consequently, any decision that R_{FB} can propose must have been issued by a client (and is valid). Correct replicas can thus only receive valid decisions. Because any majority must involve correct replicas, it follows by induction that for all future views and respective fallback leaders, any (valid) proposed decision must have been proposed by a client.

Note that if R_{FB} is Byzantine, it may collect two ElectFb message quorums with different majorities, and equivocate by sending different (valid) DecFb messages to different replicas. In this case, different correct replicas may not adopt the same decision (thus precluding the generation of a decision-certificate). Nonetheless, both decisions are valid as they must have originally been proposed by some client. □

We conclude our proof:

Theorem 3 *Invoking the fallback mechanism preserves Theorem 1 and Theorem 2.*

Proof. Lemma 5 states that a fallback leader can only propose decisions that were proposed by clients. Lemma 4 additionally guarantees that once slow path c-cert/a-cert exist, the fallback mechanism cannot change them. It follows that Lemma 2 continues to hold. Consequently, since any c-cert/a-cert generated through the fallback mechanism are indistinguishable from normal case operation, Theorems 1 and 2 remain valid. □

A Note on Matching Views Slow path c-cert's and a-cert's require matching st2r messages with matching decision views. This is necessary for safety in case no decision has been successfully logged so far. Consider an example in which for a view v half of the replicas have adopted the decisions Commit and Abort respectively. Let P_1 (Commit) and P_2 (Abort) be the partitions of replicas with matching votes. A Byzantine fallback leader may equivocate and form two ElectFb quorums with different majorities, and send dec_{v+1} = Abort to P_1, and dec_{v+1} = Commit to P_2. If non-matching views were allowed, a c-cert could exist by using P_1's st2r's from view v and P_2 st2r's from view $v + 1$, and vice versa for the a-cert, violating Lemma 2. ElectFb quorums, however, need not match in views for safety. This follows from Lemma 4: once there exists a c-cert (or analogously an a-cert) for view v no correct replica can ever vote for a different decision in any future view $v' > v$. Any ElectFb quroum consequently will contain a majority for said decision.

3.5.4 Fallback Liveness

We first show that, during sufficiently long synchronous periods, the election of a correct fallback leader succeeds. Concretely, we say that after some unknown *global synchronization time* (GST), an upper bound Δ holds for all message delays.

In practice, replicas enforce exponential timeouts on each new view: a replica will not adopt a new view and start a new election until the previous view leader (whether client(s) or fallback replica) has elapsed its timeout. Replicas may, additionally, allot an initial grace period timeout for the origniating client (view 0) to avoid unnecessary fallback invocation and interference from concurrent interested clients.

For convenience, we re-iterate the view change rules (§ 3.4, protocol step 2). A replica that receives an InvokeFb message updates its current view as follows.

- **R1**: If a view v appears at least $3f + 1$ times among the *current views* included in InvokeFb, and v is larger than the replicas current view, the replica adopts a new current view $v_{new} = v + 1$.
- **R2**: Otherwise, the replica sets its current view to the maximum of its current view and the largest view that appears at least $f + 1$ times among the *current views* included in InvokeFb.

When counting how frequently a view is present in *current views*, R uses vote subsumption: the presence of view v counts as a vote also for all $v' \leq v$.

We first show that a majority of correct replicas do not diverge in their current view by more than one, allowing all correct replicas to quickly synchronize.

Lemma 6 *At any time there are at least* $2f + 1$ *correct replicas that are at most one view apart.*

Proof. A client must provide $3f + 1$ matching *current views* in the InvokeFb message (rule **R1**) in order for replicas to adopt the next view and send an ElectFb message. This implies that if any correct replica is currently in view v, there must exist at least $2f + 1$ correct replicas in a view no smaller than $v - 1$. □

Theorem 4 *A correct client can reconcile correct replicas' views in at most two round-trips and one timeout.*

Proof. Let v be the highest current view held by any correct replica. Since there are $n = 5f + 1$ total replicas, an interested client trying to InvokeFb can wait for at least $n - f = 4f + 1$ replica replies. If the client receives $3f + 1$ matching views for a view $v' \geq v - 1$ it can use **R1** to propose a new view $v'' = v' + 1 \geq v$ that will be accepted by all correct replicas in just a single round-trip.

If a client cannot receive such a quorum, *e.g.*, due to temporary view inconsistency, it must reconcile the views first. This is possible in a single additional step: By Lemma 6, any set of $4f + 1$ replica responses must contain at least $f + 1$ correct replicas' votes for a view $v' \geq v - 1$. Using **R2**, all correct replicas lagging behind may skip ahead to v'. Thus, in a second round, the client will be able to receive $\geq 4f + 1$ replica replies for a view $\geq v'$, enough to apply **R1** and move all replicas to a common view $\geq v \geq v' + 1$. We point out that while only two round-trips of message delays are required, a client may still have to wait out the view timeout between $v - 1$ and v (where v is the highest view held by any correct replica).

It follows that a correct client requires at most two round-trips and one timeout to bring all correct replicas to the same view. □

<u>Note</u>: If only a few replicas are in the highest view (*i.e.*, less than $f + 1$— otherwise we could potentially catch up in a single round-trip using **R2**), and $4f + 1$ replicas are in the same view after using **R2**, then a client does not even need another roundtrip (and a timeout) in order to guarantee the successful leader election (*e.g.*, if Byzantine replicas behave correctly there may be $4f + 1$ replicas that send an ElectFb message).

Next, we show that during sufficiently long periods of synchrony, a correct fallback leader is eventually elected.

Lemma 7 *After GST, and in the presence of correct interested clients, a correct fallback leader is eventually elected.*

Proof. In the presence of a correct interested client, Byzantine clients cannot stop the successful election of a new fallback leader (*e.g.*, by continuously invoking a view-change on only a subset of replicas in an effort to keep views divergent). This follows directly from the fact that after GST, once the timeouts for views grow large enough to fall within Δ, a correct client will (*i*) bring all correct replicas to the same view, and (*ii*) there is sufficient time for the fallback replica to propose a decision before replicas move to the next view (and consequently reject any proposal from a lower view).

Finally, we note that election does not skip views as a correct client will broadcast a new-view invocation to all replicas.

Since fallback leader election is round-robin, it follows that a correct fallback replica will be elected after at most $f + 1$ view changes, and that its tenure will be sufficiently long to reconcile a decision across all correct replicas. $\square$

Next, we show that Lemma 7 allows correct clients to complete any transaction.

Theorem 5 *A correct client eventually succeeds in acquiring either a* C-CERT *or* A-CERT *for any transaction of interest.*

Proof. First, we note that a timely client can trivially complete all of its own transactions that have no dependencies. However, if a client is slow, or its transaction has dependencies, it may lose autonomy over its own transaction. For a given client c, we define the set $Interested_c$ to include its own transactions and all of its transaction dependencies, as well as any other arbitrary transactions whose completion a client is interested in.[3]

We distinguish two cases for each transaction $T \in Interested_c$ whose original client has failed to make timely progress:

(*i*) An interested client *c* manages to receive a C-CERT/A-CERT by either issuing a RP message and receiving a Fast-Path Threshold of ST1R messages for all involved shards, or by issuing a new ST2 message and receiving $n - f$ ST2R messages from the logging

[3] A client may, for instance, drive completion of unfinished transactions for altruistic reasons (*e.g.*, garbage collection).

shard. In this case, a client is able to complete the transaction independently as any client may broadcast C-CERT's/A-CERT's to the involved shards. Theorems 1 and 2 continue to hold as this case follows the normal-case protocol operation.

(*ii*) An interested client cannot obtain decision certificates and invokes the fallback protocol. By Lemma 7, during a sufficiently long synchronous period, a correct fallback leader will be elected after at most $f + 1$ views. This fallback leader will reconcile a consistent decision across all correct replicas in the logging shard, thus allowing the interested client c to receive a $n - f$ matching ST2R messages. This allows c to construct the respective C-CERT/A-CERT and proceed to the Writeback Phase. □

Practical Optimizations

The first view A replica in view 0 can accept an InvokeFb message without a proof of replica *views*, allowing a client to propose view 1 without first collecting a set of signed current views. Since replicas begin in view 0 and only increment their view without proof once, this optimization trivially maintains the invariant that a majority of correct replicas are no more than one view apart (Lemma 6). In synchronous, fault-free settings, only a single fallback invocation is necessary, allowing for transaction recovery without additional view signatures.

Bounding divergence In the absence of correct interested clients, Byzantine clients may invoke fallback elections at a subset of replicas, either in an effort to move replicas to large views (with high timeouts) or to selectively skip candidate leaders. To avoid this, replicas may opt to share their ElectFb messages to all other replicas; ElectFb messages addressed to non-leader replicas (for a given view) need not be signed, but may simply use MACs. Replicas that receive $f + 1$ ElectFb messages adopt the view (if larger than their current) and create an ElectFb message themselves, thus ensuring that all correct replicas adopt each new view, and a fallback leader is successfully endorsed.

This optimization is not necessary for liveness but can speed up recovery once a correct client becomes interested. To limit all-to-all communication, one might choose to only enable the optimization for views $v > t$, where t can be a system hyperparameter. If the additional communication, instead, is of no concern, this optimization may also be used *instead of* client-driven InvokeFb messages in an effort to reduce signature costs. Clients can, for instance, simply *request* a view change: replicas enter the next view and broadcast a new ElectFb message once they have (*i*) received a request, (*ii*) elapsed their current view timeout, and (*iii*) received at least $2f + 1$ ElectFb messages for the current view (enough to guarantee every correct replica will receive at least $f+1$ ElectFb messages and join the current view). This ensures that correct replicas neither diverge too far, nor skip views.

3.5.5 Revisiting Vote Subsumption

Our discussion of the recovery protocol thus far has relied on the use of *vote subsumption* for validating and choosing a new current view when invoking a fallback leader election. Vote subsumption allows a client to form an InvokeFb message using a set of *current views* that includes replica signatures cast for non-matching views: when counting how frequently a view is present in the received *current views* a replica considers the presence of view v as a vote also for all views $v' \leq v$.

This simplifies the election process, but limits the applicability of signature aggregation schemes. While some schemes [26, 69] support the aggregation of signatures on non-matching messages, more efficient *multi-signature* [24, 25, 90, 129] or *threshold* *signature* [24, 29, 163] schemes require matching messages.

In the following we briefly show that vote subsumption is not necessary for progress, and how to modify Basil's fallback protocol to omit it if desired. We show that client can, by reasoning carefully about the set of current replica views received, always wait for matching responses.

Lemma 8 *An interested client trying to send* InvokeFb *can always use matching current views.*

Proof. Since there are $n = 5f + 1$ total replicas an interested client trying to submit an InvokeFb can wait for at least $4f + 1$ current view messages from different replicas. Let view v be the largest view among the received views.

If v was sent by a correct replica, then it follows from Lemma 6 that there must be at least f other views $v' \geq v - 1$ among the received views. If there are not, then the client can conclude that v must have been sent by a Byzantine replica and hence it can remove v (and the replica that sent it) from consideration and wait for an additional current view message.

We assume in the remainder that v is the largest observed view that meets the above criterion. Thus, the client received at least $f + 1$ current views that are at most one view apart from one another (v and v'). We distinguish three cases:

① There are $3f + 1$ matching views: The client can use rule **R1** to send an InvokeFb message using only matching views. □

② There are $f + 1$ matching views: The client can use rule **R2** to send an InvokeFb message using only matching views. □

③ There are fewer than $f + 1$ matching views. However, as established above, there are at least $f + 1$ current views (in particular two, v and v') that are at most one view apart from one another. We distinguish two sub-cases:

- $\underline{v \text{ was sent by a correct replica:}}$ Then it follows from Lemma 6 that at least $2f + 1$ total correct replicas must be in a view $v'' \geq v - 1$. Thus, if the client does not observe at least $f + 1$ matching current view messages for either view v or v', then it can keep waiting. By the pigeonhole principle, it must receive $f + 1$ matching current views for v or v'.
- $\underline{v \text{ was not sent by a correct replica:}}$ Then it follows that v' must have been sent by a correct replica, since $f + 1$ replicas reported either v or v' as their current view. It follows from Lemma 6 that there are at least $2f + 1$ correct replicas in view $v'' \geq v' - 1$. Consequently the client can keep waiting until it observe either $f + 1$ matching current views for v' or for v''.

In both cases the client is able to wait for $f + 1$ matching current views, and hence it can use **R2** to send an InvokeFb message using only matching views. □

Since the client can always use matching messages, it follows that both multi-signature and threshold signature schemes are applicable to InvokeFb messages.

Lemma 8 does not suffice to conclude progress since we only showed that a client can always succeed in receiving enough matching replies to apply rule **R2**. However, it follows directly that a client will also be able to apply rule **R1** within at most another exchange, and thus ensure progress for continued fallback leader election.

We conclude:

Theorem 6 *Vote subsumption is not necessary for progress in Basil.*

Proof. Since clients are guaranteed to gather sufficient catch up messages (Lemma 8) it is guaranteed that, after catching up, there will be at least $4f+1$ correct replicas within at most one view of each other. Let v_{max} be the larger of the two views. By the pigeonhole principle, a client is guaranteed to be able to wait for least $f+1$ matching votes for v_{max} or $3f+1$ matching votes for $v_{max}-1$. Thus, the client can move all replicas to view v_{max} (using only matching views), ensuring progress. □

A correct client can ensure progress (successful leader election) during a "stable" timeout (after GST) in which replicas are not changing views while the client is waiting for replies. If replicas are changing their views, then waiting for the first received message from a replica is potentially insufficient to receive matching replies: A client can (and needs to) keep waiting until it *does* receive matching messages, potentially displacing old replica votes with newer ones. Eventually, given partial synchrony, there will be a long enough timeout to ensure stability and progress. Replicas can simply send their new view to each interested client everytime they increment it or rely on periodic client pings to query their current view.

3.6 Evaluation

Our evaluation seeks to answer the following questions:

- How does Basil perform on realistic applications? (§ 3.6.1)
- Where do Basil's overheads come from? (§ 3.6.2)
- What are the impacts of our optimizations in Basil? (§ 3.6.3)
- How does Basil perform under failures? (§ 3.6.4)

Baselines We compare against three baselines: (*i*) TAPIR [195], a recent distributed database that combines replication and cross-shard coordination for greater performance but does not support Byzantine faults; (*ii*) TxHotstuff, a distributed transaction layer built on top of the standard C++ implementation [191] of HotStuff, a recent consensus protocol that forms the basis of several commercial systems [10, 16, 31, 179, 181], most notably Facebook Diem's Libra Blockchain; and (*iii*) TxBFTsmart, a distributed transaction layer built on top of BFT-SMaRt [23, 176], a state-of-the-art PBFT-based implementation of Byzantine state machine replication (SMR).[4]

HotStuff and BFT-SMaRt support general-purpose SMR, and are not fully-fledged transactional systems; we thus supplement their core consensus logic with a coordination layer for sharding—running the Two-Phase Commit (2PC) protocol—and an execution layer that implements a standard optimistic concurrency control serializability check [104] and maintains the underlying key-value store. This architecture follows the standard approach to designing distributed databases (*e.g.*, Google Spanner [40], Hyperledger Fabric [8] or Callinicos [145, 146]) where concurrency control and 2PC are layered on top of the consensus mechanism. Spanner and Hyperledger (built on Paxos and Raft, respectively) are not Byzantine-tolerant, while Callinicos does not support interactive transactions. To the best of our knowledge, ours is the first academic evaluation of HotStuff as a component of a transactional system.[5]

We use ed25519 elliptic-curve digital signatures [19, 134] for both Basil and the transaction layer of TxHotstuff and TxBFTsmart. Additionally, we augmented both BFT baselines to also profit from Basil' reply batching scheme.

Experimental Setup We use CloudLab [36] `m510` machines (8-core 2.0 GHz CPU, 64 GB RAM, 10 GB NIC, 0.15 ms ping latency) and run experiments for 90 seconds (30 s

[4]Our system prototypes can be found at `https://github.com/fsuri/Basil_SOSP21_artifact`.
[5]We discussed extensively our setup and implementation with the authors of TAPIR and HotStuff.

warm-up/cool-down). Clients execute in a closed-loop, reissuing aborted transactions using a standard exponential backoff scheme. We measure the latency of a transaction as the difference between the time the client first invokes a transaction to the time the client is notified that the transaction committed. Each system is configured to tolerate $f = 1$ faults ($n = 2f + 1$ for TAPIR, $n = 3f + 1$ for HotStuff and BFT-SMaRt, and $n = 5f + 1$ for Basil).

3.6.1 High-level Performance

We first evaluate Basil against three popular benchmark OLTP applications: TPC-C [182], Smallbank [50], and the Retwis-based transactional workload used to evaluate TAPIR [195]. TPC-C simulates the business logic of an e-commerce framework. We configure it to run with 20 warehouses. As we do not support secondary indices, we create a separate table to (*i*) locate a customer's latest order in the `order status` transaction and (*ii*) lookup customers by last name in the `order status` and `payment` transactions [42, 167]. We configure Smallbank, a simple banking application benchmark, with one million accounts. Access is skewed, with 1,000 accounts being accessed 90% of the time. Users in the Retwis-based benchmark, which emulates a simple social network, similarly follow a moderately skewed Zipfian distribution (0.75). Figures 3.5 and 3.6 report results for the three applications.

TPC-C Basil's TPC-C throughput is 5.2x higher than TxHotstuff's and 3.8x higher than TxBFTsmart's—but 4.1x lower than TAPIR's. All four systems are contention-bottlenecked on the read-write conflict between `payment` and `new-order`. Basil has 4.2x higher latency than TAPIR: this increases the conflict window of contending transactions, and thus the probability of aborts. Basil's higher latency stems from (*i*) signature

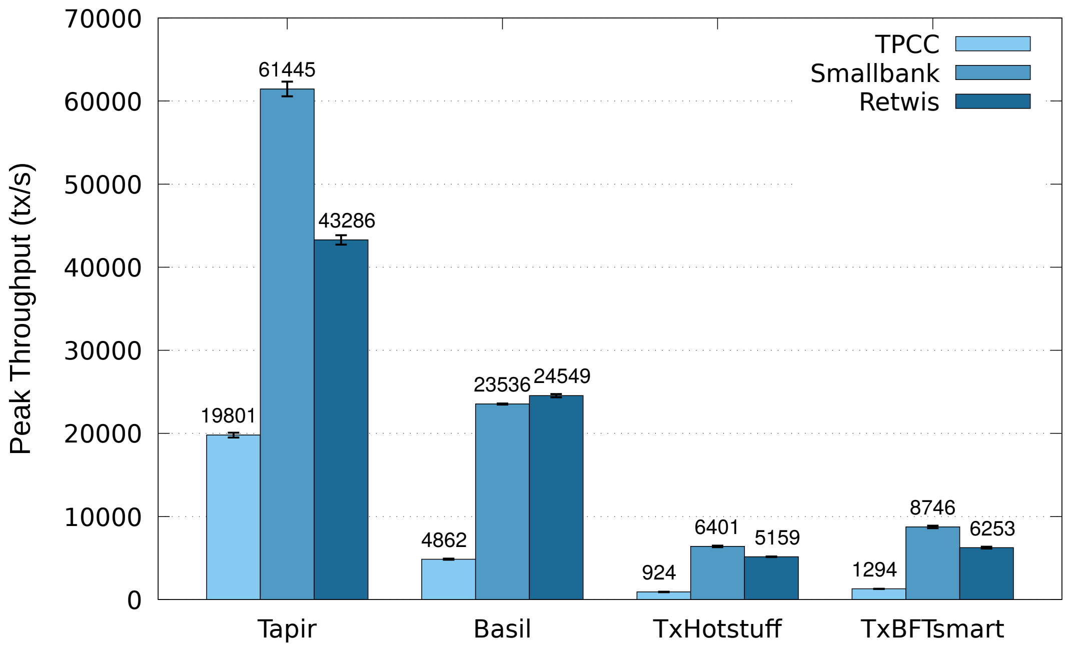

Figure 3.5: High-level performance—transaction throughput in tx/s

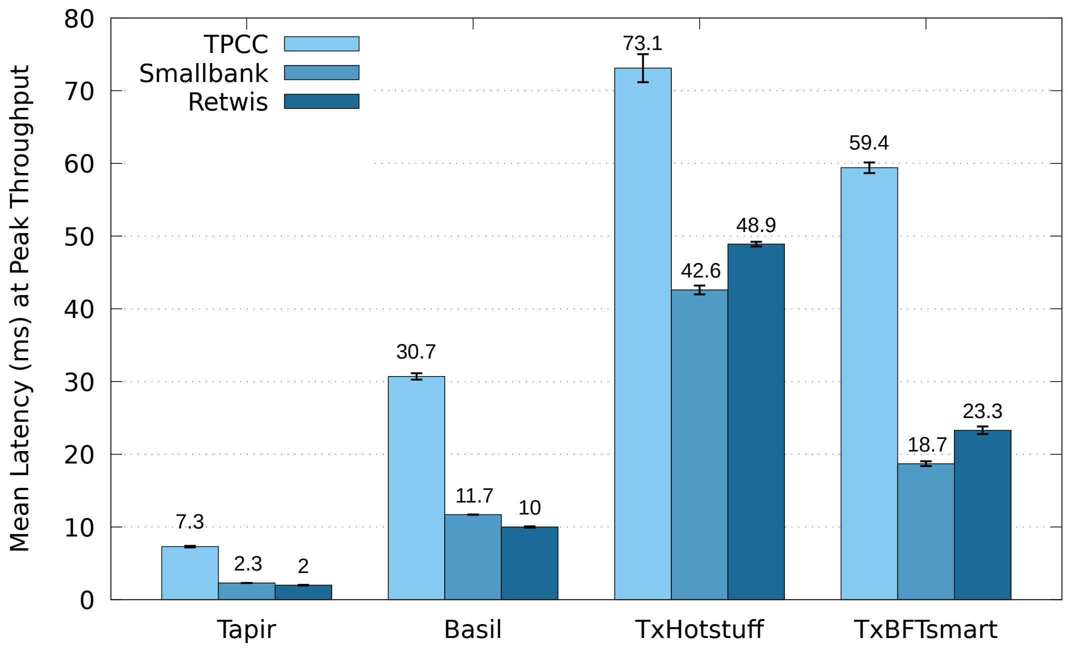

Figure 3.6: High-level performance—transaction latency in ms

generation and verification costs at replicas; (*ii*) its larger quorum sizes for both read and prepare phases; and (*iii*) its need to validate read/prepare replies at clients.

Throughput in Basil is higher than in TxHotStuff and TxBFTsmart. Basil's superior performance is directly linked to its lower latency (2.4x lower than TxHotstuff, 1.2x lower than TxBFTsmart). By merging 2PC with agreement, Basil allows transactions to decide whether to commit or abort in a single round-trip 96% of the time (through its fast path). TxHotstuff and TxBFTsmart, which layer a 2PC protocol over a black-box consensus instance, must instead process and order two requests for each decision (one to Prepare and another to Commit/Abort), each requiring multiple roundtrips. In particular, Hotstuff and BFT-SMaRT incur nine and five message delays, respectively, before returning the Prepare result to clients. In a contention-heavy application like TPC-C, this higher latency translates directly into lower throughput, since it significantly increases the chances that transactions will conflict. Indeed, for these applications, layering transaction processing on top of state machine replication actually turns a classic performance booster for state machine replication—running agreement on large batches—into a liability. Large batches increase latency and encourage clients to operate in lock-step, increasing contention artificially. In practice, we find that TxHotstuff and TxBFTsmart perform best with comparatively small batches (four transactions for TxHotStuff, and 16 transactions for TxBFTsmart).

Smallbank and Retwis Basil is only 1.8x and 2.6x slower than TAPIR for the respective workloads, which are resource bottlenecked in both systems. The lower contention in Smallbank and Retwis (due to the relatively small transactions) allows Basil to use a batch size of 16 for signature generation (up from 4 in TPC-C), thus lowering the cryptographic overhead that Basil pays over TAPIR. With this larger batch size, both TAPIR and Basil are bottlenecked on message serialization/deserialization and networking overheads. Because of their higher latency, however, TxHotStuff and TxBFTsmart continue to be contention bottlenecked: Basil's commit rates for Smallbank and Retwis are 93% and 98%, respectively, but for TxHotStuff they drop to 75% and 85%, respectively, and for TxBFTsmart to 85% for both benchmarks. Even on their best configuration (a batch size of 16 for TxHotStuff and 64 for TxBFTsmart), Basil outperforms them, respectively, by 3.7x and 2.7x on Smallbank, and by 4.8x and 3.9x on Retwis.

3.6.2 BFT Overheads

Besides requiring additional replicas (from $2f+1$ in TAPIR to $5f+1$ in Basil), tolerance to Byzantine faults requires both additional cryptography to preserve Byz-serializability and more larger read quorums to preserve Byzantine independence. To evaluate these overheads, we configure the YCSB-T microbenchmark suite [39] to implement a simple workload of identical transactions over ten million keys. We distinguish between a uniform workload (*RW-U*) and a Zipfian workload (*RW-Z*) with coefficient 0.9.

We first quantify the cost of cryptography. To do so, we measure the relative throughput of Basil with and without signatures. When using signatures, we use a batch size of 16 for *RW-U* and 4 for *RW-Z.* Transactions consist of two reads and two writes. Figure 3.7 describes our results. We find that Basil without cryptography performs 3.7x better than Basil with cryptography on the uniform workload (*RW-U*), and up to 4.6x better on the skewed workload (*RW-Z*). Without cryptography, Basil can use cores that would have been dedicated for signing/signature verification for regular operation processing. This effect is more pronounced on the skewed workload as reducing latency (through increased operation parallelism, lack of batching, and absence of signing/verification latency) reduces contention, and thus further increases throughput.

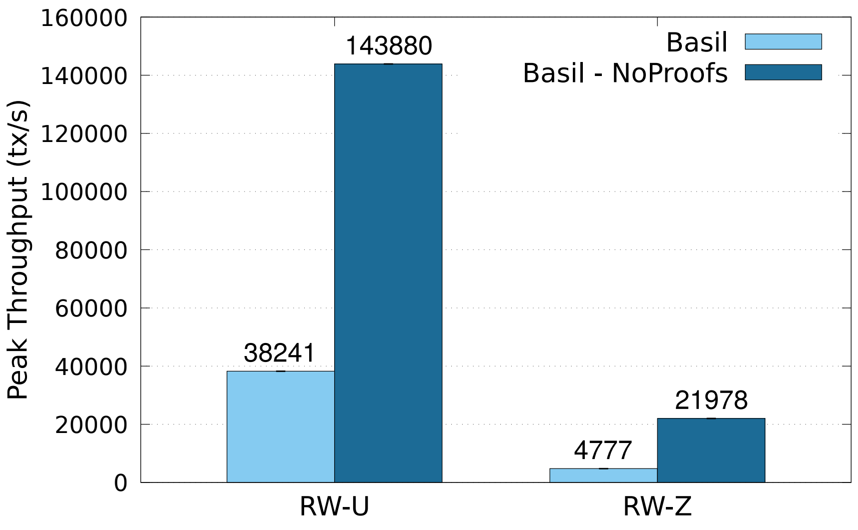

Figure 3.7: Performance impact of signatures

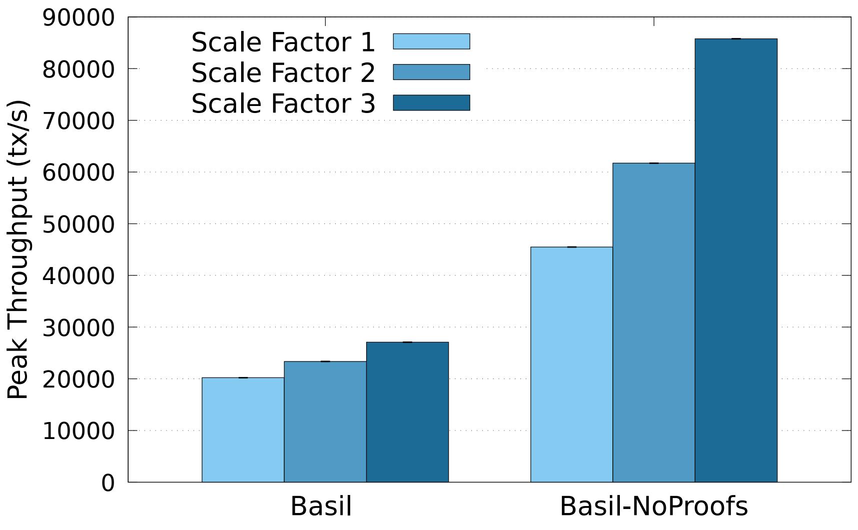

Figure 3.8: Throughput scalability with an increasing number of shards

In all sharded BFT systems, the number of signatures necessary per transaction grows linearly with the number of shards: each replica must verify that other shards also voted to commit/abort a transaction before finalizing the transaction decision locally. This requires signed votes from each shard. In Figure 3.8, we quantify this cost by increasing the number of shards from one to three on the CPU-bottlenecked *RW-U* workload. We configure transactions to consist of three reads and three writes. Each shard has its own cluster of replica machines, and we run the system with a batch size of 16. Basil without cryptography increases by a factor of 1.9x (on average, transactions with three read operations will touch two distinct shards). In contrast, Basil's throughput increases by only 1.3x.

To guarantee Byzantine independence, individual clients must receive responses from at least $f + 1$ replicas (TAPIR clients, instead, can read from single replica). Reading from additional replicas (at least $2f + 1$) can improve freshness—it may increase the chances that a transaction successfully reads the latest prepared write (acquiring a valid dependency requires $f + 1$ matching prepared versions)—at the cost of diminished load balancing. We measure the relative cost of different read quorum sizes in Figure 3.9. We use a simple read-only workload consisting of 24 read operations per transaction, and use a batch size of 16. Unsurprisingly, increasing the read quorum (*i*) increases the load on each replica—each replica must process additional messages and generate signatures for replies—, and (*ii*) increases verification costs for clients—more replica signatures (and C-CERT's) must be processed per read. Throughput decreases by 20% when reading from $f + 1$ replicas (instead of one), and a further 16% when reading from $2f + 1$.

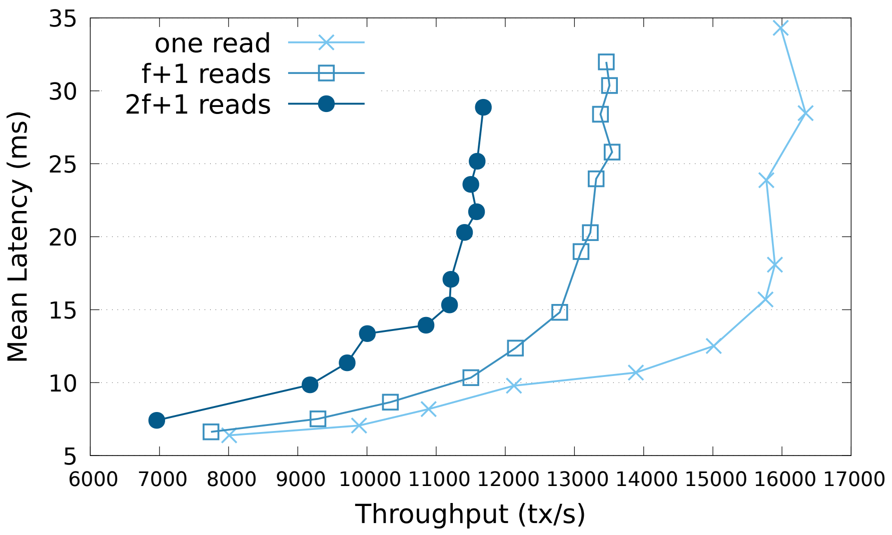

Figure 3.9: Performance impact of different read quorum sizes

3.6.3 Basil Optimizations

We measure how Basil's performance benefits from batching and from its fast path option. We report results for YCBS-T with and without fast path (NoFP) on a workload of two reads and two writes (Figure 3.10). For the uniform workload (*RW-U*), enabling fast paths leads to a 19% performance increase; the ST2R messages that fast paths save contain a signature that must be verified, but require little additional processing. For a contended Zipfian workload (*RW-Z*), however, the additional phase incurred by the slow path increases contention (as it increases latency): adding the fast path increases throughput by 49%. Note that Byzantine replicas, by refusing to vote or voting abort, can effectively disable the fast path option; Basil can try to prevent this by removing consistently uncooperative replicas.

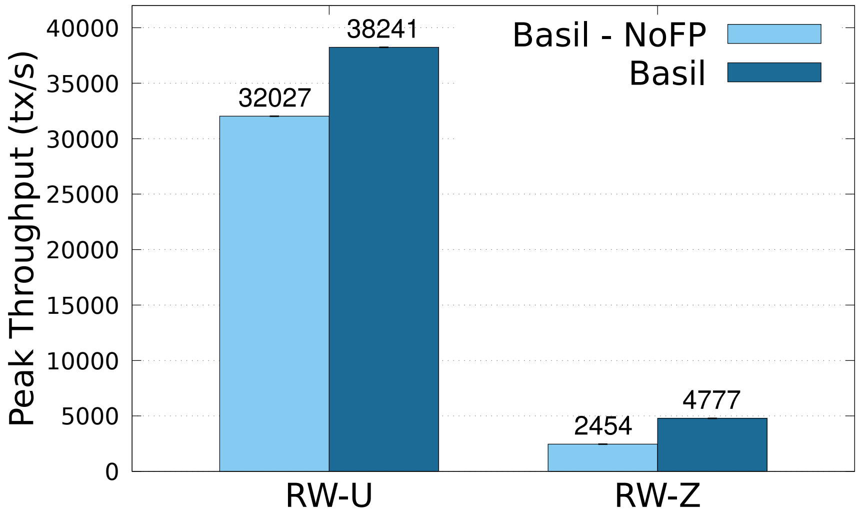

Figure 3.10: Throughput with and without the commit fast path

Next, we quantify the effects of batching. We report the throughput for both workloads (transactions again consist of two reads and two writes) while changing the batch size from 1 to 32 (Figure 3.11). As expected, on the resource-bottlenecked uniform workload (*RW-U*), throughput increases linearly with increased batch size until peaking at 16 (a 4x throughput increase)—at which point additional hashing costs of the batching Merkle tree neutralize any further reduction in signature costs. On the Zipfian workload (*RW-Z*) instead, throughput only increases by up 1.4x, peaking at a small batch size of 4, and degrading afterwards as higher wait times and batch-induced client lock step increase contention (thus reducing throughput).

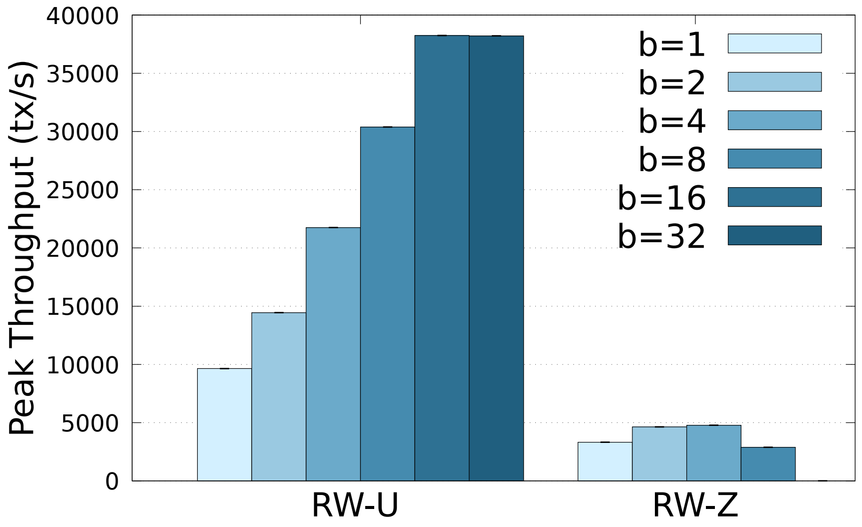

Figure 3.11: Throughput with increasing reply batch size

3.6.4 Basil Under Failures

Basil can experience Byzantine failures from both replicas and clients. We have already quantified the effect of Byzantine replicas preventing fast paths (Figure 3.10) and, by being unresponsive (or fabricating versions), forcing clients to read larger read quorums (Figure 3.9). We then focus here on quantifying the effects of client failures.

Basil clients are in charge of their own transactions. Byzantine clients can thus only disrupt honest participants when their own transactions conflict with those of correct clients. Otherwise, lack of progress impacts only themselves. A Byzantine client's best strategy for successfully disrupting execution is (*i*) to follow the (estimated/observed) workload access distribution (only contending transactions cause conflicts), (*ii*) choose conservative timestamps (only committing transactions cause conflicts) and (*iii*) delay committing a prepared transaction (forcing dependencies to block, and conflicts to abort).

Byzantine clients can stall after sending ST1 messages (*stall-early*), or before sending vote certificates V-CERT$_S$ (*stall-late*). To equivocate, they must instead receive votes that allow them to generate, and send to replicas, conflicting V-CERT certificates. We remark that equivocating, and hence triggering the divergent case recovery path, is *not* a strategy that can be pursued deterministically or even just reliably, since its effectiveness depends on the luck of the draw in the composition of ST1R's quorum. We evaluate two scenarios: a worst-case, in which we artificially allow clients to always equivocate (*equiv-forced*), and a realistic setup where clients only equivocate when the set of messages received allows them to (*equiv-real*).

For both scenarios, we report the throughput of correct clients (measured in $tx/s/client_{correct}$). We keep a constant number of clients, a fraction of which exhibits Byzantine behavior in some percentage of their newly admitted (*i.e.*, not retried) transactions; we refer to those transactions as faulty. Faulty transactions that abort because of contention are not retried, while correct transactions that abort for the same reason may need to re-execute (and hence prepare) multiple times until they commit. When measuring throughput, we report the percentage of faulty transactions as a fraction of all processed (not admitted) transactions. Figures 3.12 and 3.13 present our results.

For the RW-U workload, the additional CPU load of fallback invocations on the CPU-bottlenecked servers causes correct clients' throughput to decrease slowly and linearly. Clients invoke fallbacks only rarely, as there is no contention. Moreover, stalled transactions can be finished in a single round-trip (a pair of RP, RPR) messages thanks

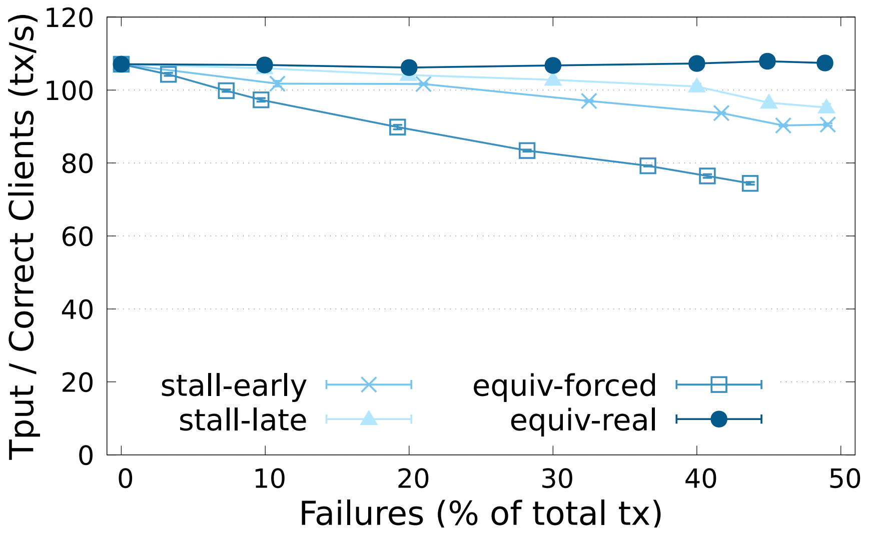

Figure 3.12: Per-client throughput of correct clients under an uncontended workload with a uniform access pattern, in the presence of client failures.

to the fallback's common case and fast path. The small throughput drop over *stall-late* is an artefact of Byzantine clients directly starting a new transaction before finishing the old one, increasing the effective input of malicious clients over correct ones. The cost of forced equivocation is higher as it requires three rounds of message processing (fallback invocation, election, and decision adoption). In reality, *equiv-real* sees no throughput drop, as the lack of contention makes equivocation impossible: Byzantine clients cannot build the necessary conflicting V-CERT's.

The RW-Z workload is instead contention-bottlenecked: higher latency implies more conflicts, and thus lower throughput. The impact of *stall-late* stalls remains small, as all affected clients still recover the transaction on the common case fast path (incurring only one extra roundtrip). The performance degradation is slightly higher in *stall-early*, as stalled-early transactions do not finish the transactions on which they depend before stalling. Instead, affected correct clients must themselves invoke the fallback for stalled dependent transactions, which increases latency. In practice, dependency chains remain small: because of the Zipfian nature of the workload, correct clients quickly

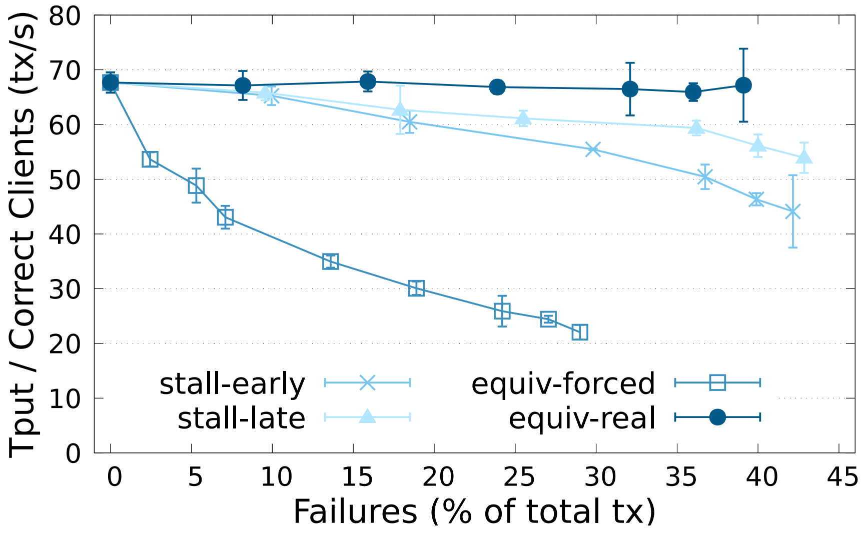

Figure 3.13: Per-client throughput of correct clients under a highly contented Zipfian workload, in the presence of client failures.

notice stalled transactions and aggressively finish them. We note that stalled transactions do not themselves increase contention: Basil allows the stalled writes of prepared but uncommitted transactions to become visible to other clients as dependencies. A stalled transaction thus causes dependency chains to grow, but does not increase the conflict window. The throughput drop that results from forcing equivocation failures is, in contrast, significant: equivocation requires three round-trip to resolve and may lead to transactions aborting and to cascading aborts in dependency chains. In practice, Byzantine clients in *equiv-real* are once again rarely successful in obtaining the conflicting ST1R messages necessary to equivocate, even in a contended workload (0.048% of the time for 40 % faulty transactions) as 99% of transactions commit or abort on the fast path.

Basil expects some level of cooperation from its participants and can remove, without prejudice, clients that frequently stall or timeout (in addition to explicitly misbehaving clients). To avoid spurious accusations towards correct clients, exclusion policies can be lenient since Basil's throughput remains robust even with high failure rates.

3.7 Extended Technical Discussion and Optional Modifications

In this section we discuss some additional technical details and optional modifications. While not integral to the core Basil protocol, these modifications can be used to improve performance, or to adapt the protocol to specific use cases; or might simply be of interest to the reader.

3.7.1 Garbage Collection

Basil is a multi-versioned system, and as such, it must be able to garbage collect old data versions.

Versions Replicas in Basil maintain a low watermark *gc*, which represents a lower bound on the set of versions actively managed. Versions with timestamps below *gc* are considered eligible for garbage collection. The *gc* watermark lags behind a replica's local time and is advanced periodically—either based on a timer, or when "too many" new writes have been applied. Replicas reject any new transaction whose timestamp $ts \leq gc$. Writes with timestamps $\leq gc$ are considered obsolete. In contrast, reads with timestamps $\leq gc$ are unsafe: if a replica has already garbage collected versions below *gc*, it may no longer be able to guarantee Byz-serializability, as relevant versions required for the MVTSO-check may have been deleted.

By default, garbage collection is performed per key, triggered by the insertion of a new version.[6] When a new version $v > gc$ is written, the replica prunes all but the latest version below *gc* for the corresponding key. The most recent version is retained to ensure that at least one readable version remains, which is necessary to support reads

[6]Garbage collection may also be performed periodically in a sweeping manner, which can be beneficial for keys that are updated infrequently.

with timestamps > gc. Retaining the latest version also ensures that serializability can still be enforced for readers that may have observed a now-deleted version. Without this, such read transactions could no longer be definitively checked for conflicts.

Once a version is garbage collected, the replica also deletes its associated transaction object and metadata (*e.g.*, commit certificates). This is safe because the transaction object is no longer needed to service reads or writes—replicas ignore any requests whose timestamp is below gc.

For auditing purposes, replicas may optionally archive garbage-collected versions and their metadata to external storage. For example, an offline auditing service could inspect all transactions and their read/write sets to verify client behavior—such as ensuring clients did not read or write unauthorized values, or violate application semantics.

Decision Certificates In addition to transaction objects and their associated decision metadata, replicas also maintain decision certificates (C-CERT/A-CERT). These certificates serve multiple purposes: (*i*) they assert the validity of a supplied read version (the C-CERT proves the version committed), (*ii*) they serve to propagate Writeback decisions that assist interested clients during the fallback protocol, and (*iii*) they enable an abort fast path for validation by providing conflict proofs (the C-CERT proves the conflicting transaction committed).

Decision certificates are garbage-collected alongside their corresponding transactions. However, because certificates may be comprised of large signature quorums, it can be desirable to collect them more aggressively. In principle, certificates can be safely discarded whenever, since they are not required to preserve serializability: the existence of a certificate implies that a corresponding decision has been durably logged on at least $2f + 1$ correct replicas—sufficient to preclude committing conflicting transactions.

When a replica deletes a certificate, it retains only the decision. It can partake in fallback invocations by voting with a FINAL decision: a quorum of $f + 1$ matching FINAL decisions is sufficient to serve as a Writeback certificate. If such a quorum is unavailable, the vote may instead be used as a ST2R message. To enhance the robustness of the fallback process—particuarly to avoid involving all shards during resolution—a replica may choose to retain the certificate until it has confidence that at least $2f + 1$ correct replicas have finalized the decision. For instance, replicas can gossip finalized decisions or rely on an off-critical-path Writeback acknowledgement round involving the client. This ensures that every interested client receives at least $f + 1$ matching FINAL commit decisions.

Alternatively, resolution can proceed by confirming that $2f + 1$ correct replicas have cast matching ST2R messages. (A FINAL decision does not necessarily imply the existence of ST2R messages—for example, when a transaction completes via the fast path.) In this case, the decision can be resolved entirely on the logging shard. Crucially, any quorum of $n - f$ messages observed by a client will contain at least $f + 1$ matching decisions—sufficient for other replicas to validate and adopt the decision, even if they have not yet received any ST2 messages.

3.7.2 Optimizing Transaction Objects

Hierarchical Transaction IDs Transactions in Basil may span multiple shards but, by default, store a single global *ReadSet*, *WriteSet*, and *DependencySet*. Although each shard only applies writes and performs validation (via the MVTSO-check) for keys in its own namespace, it still stores the entire transaction object. This leads to unnecessary storage overhead, as every shard effectively stores (meta-)data for the entire database.

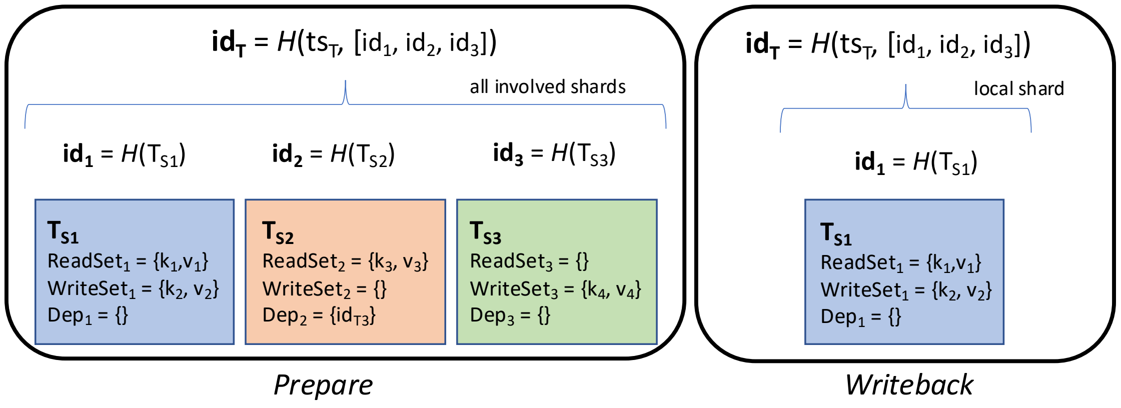

Figure 3.14: Revised transaction object structure using hierarchical identifiers. Read and write set metadata is partitioned by shard, with each shard retaining only its locally relevant portion upon commit.

To address this inefficiency, Basil partitions the transaction object into per-shard sub-objects (Fig. 3.14). Each sub-object T_{S_i} is specific to shard S_i and is assigned a unique identifier $id_i = H(T_{S_i})$, computed as the hash of the sub-object. The global transaction identifier $id_T := H([id_i, ..., id_j], ts_T)$ is then computed as the hash of all the collection of all sub-identifiers and the transaction timestamp ts_T.

During the *Prepare* phase, the client sends the complete transaction object—including all sub-objects—to each involved shard. This redundancy ensures that any shard can aid in recovery if a malicious client never sends the transaction to every involved shard. Each replica in shard S_i verifies that: (*i*) T contains a sub-object for each involved shard, (*ii*) *all* T_{S_j} match their claimed digest id_j, and (*iii*) the global identifier id_T is correct. It then runss the MVTSO-check exclusively on its local sub-object T_{S_i}. If T_{S_i} contains a key outside S_i's partition, this serves as proof of client misbehavior; the request is rejected, and then the client may be blacklisted.

During the *Writeback* phase, clients need only send the relevant sub-object to each shard, *e.g.*, T_{S_i} to shard S_i. Likewise, shards may discard non-local portions of a previously stored transaction object. Since a valid commit Writeback message must include a shard-level decision for each involved shard, it follows that all shard have durably replicated their respective sub-objects. As a result, each shard no longer needs to retain the global transaction object.

Efficient Write Certificates For read operations, Basil relies on *commit certificates* (C-CERT) to validate the correctness of a returned (committed) write version. A C-CERT serves as a proof of commitment: it contains a quorum of signatures over the transaction's digest (id_T) and its decision. To validate a commit read, a replica must return: (*i*) the (latest) committed version of the key, (*ii*) the corresponding transaction's C-CERT, and (*iii*) the full transaction object. The inclusion of the full transaction object is necessary to verify that the returned version originated from one of the transaction's writes, and that the transaction indeed committed. However, sharing the entire transaction object can be costly—especially for large transactions—and is often unnecessary.

To mitigate this overhead, we adopt a hierarchical organization for the transaction, similar to the optimization discussed for per-shard transaction sub-objects. These two optimizations are orthogonal; for simplicity we assume a single-shard transaction in this section, though the same technique can be independently applied to each shard-specific sub-object.

We further deepen the transaction object's hierarchy by structually encoding the WriteSet itself. Rather than compute the transaction identifier as a flat digest of the entire ReadSet and WriteSet, we compute it as the digest of their respective sub-digests: $id_T = H([d(ReadSet), d(WriteSet)], ts_T)$. Here, the WriteSet sub-digest is computed as the *Merkle root* σ [128] of the WriteSet, where the leaves are key-value pairs. This hierarchical encoding enables efficient validation of individual writes without requiring access to the entire transaction object. Speficially, a client only needs to receive: (*i*) the write version w, (*ii*) the corresponding C-CERT, and (*iii*) a *write certificate*, consisting of the transaction timestamp ts_T, the ReadSet digest $d(ReadSet)$, the WriteSet Merkle root σ, and the Merkle proof π associated with w. The client can then verify that the write version w is indeed part of the transaction's WriteSet by checking that $\sigma = M(\pi, w)$, where M denotes Merkle tree verification. Finally, the client recomputes the transaction identifier $id_T = H([d(ReadSet), d(WriteSet)], ts_T)$ and verifies that it matches the one signed in the C-CERT.

3.7.3 Aiding Write Commitment

By default, Basil prioritizes validation success for readers—for example, through the use of Read Timestamps (RTS) and by selecting transaction timestamps at the beginning of execution. This is often a reasonable default: reads require a round-trip to execute, whereas writes can be buffered locally. However, this approach can increase the likelihood of aborts for long-running read-write transactions, whose writes may become stale by the time they reach validation. In this section, we discuss a few optional modifications that aim to improve success rates for writers. We note that this represents an inherent trade-off: improving outcomes for writers often comes at the expense of readers, and vice versa.

Late Timestamp Selection In Basil, transaction timestamps are selected at transaction start, and thus may be "old" by the time the transaction completes. While this poses no issue for readers—who operate on a consistent snapshot in time—it can negatively impact writers. For example, a write issued by a long-running transaction might conflict with a more recent read from a shorter, already-committed transaction, simply because its timestamp is outdated.

One way to favor writers it to select fresher timestamps. This can be achieved by either (*i*) assigning the transaction timestamp only at commit time (*e.g.*, as the maximum of all read versions and the local time), or (*ii*) progressively updating the timestamp during execution, based on observed versions or elapsed time. These strategies make writers appear more current, aligning validation more closely with their actual commit time. However, this comes at a cost: readers may now see a larger conflict window—that is, a greater delta between the observed version and the transaction's final timestamp—leading to higher abort rates for readers.

Retries An alternative, more dynamic approach is to allow transactions that fail validation to *retry* the Prepare phase with a new timestamp [195]. When a replica votes to abort a write due to a read-write conflict (Alg. 1, lines 9-10), it may return a special (VOTE-RETRY) message, along with a suggested retry timestamp (greater than the conflicting read version).

Functionally, a retry vote is equivalent to an abort vote (VOTE-ABORT) and counts the same in vote tallies. However, a client that fails to collect enough commit votes to form a *CommitQuorum* (*CQ*), but receives at least $|\text{VOTE-COMMIT}| + |\text{VOTE-RETRY}| \geq 3f + 1$ votes in total, may opt to retry its transaction commit with a higher timestamp. The retry timestamp is typically the maximum among the suggestions in the VOTE-RETRY messages, capped by a configurable bound to prevent Byzantine manipulation. This mechanism ensures that at least $3f + 1$ replicas view the transaction as either valid in its current form or acceptable after timestamp adjustment, providing a path to commit without full re-execution.

That said, retrying with a new timestamp is not a panacea. In many cases, an abort decision will persist across retries: selecting a higher timestamp with the same ReadSet may introduce new conflicts with concurrent transactions. In such cases, aborting and re-executing is the only viable path to success. For this reason, clients are encouraged to enforce a configurable retry policy, which may include both a maximum number of retry attempts and an upper bound on the allowed *conflict window*—defined as the delta between the smallest read version and the selected transaction timestamps.

Importantly, in Basil, transaction identifiers id_T are uniquely defined by the transaction timestamp. As a result, retrying a transaction results in a physically distinct transaction. To avoid unintentionally committing the logical transaction twice—at different timestamps, which could lead to anomalies—a correct client should wait until the original transaction has durably logged an abort decision before issuing a retry. In practice, the retry's Prepare phase can be piggybacked on the Writeback message that records the abort of the original attempt.

Byzantine clients, of course, are not bound by this, and may submit multiple logically identical transactions with different timestamps—whether as retries or as separate submissions. While this behavior is tolerated by the protocol—since each transaction is uniquely identified by its timestamp and validated independently—it may violate application-level semantics. As such, an external auditing layer may interpret this behavior as misbehavior, depending on the application's correctness criteria.

Read Leases Read Timestamps (RTS) allow readers to tentatively register commit intent *during* execution, increasing their chances of committing successfully: concurrent writers that attempt to prepare conflicting writes at lower timestamps are aborted instead. RTS are not necessary for correctness—they are a performance optimization that improves commit success for headers in highly concurrent workloads. RTS are also, by design, transient. They affect only writes with timestamps less than the RTS. Transactions that retry with larger timestamps—either during Prepare or due to full-rexecution—are unaffected.

However, RTS can be abused by Byzantine clients that issue reads and acquire RTS without any intention of completing (and committing) a transaction. Because read-write transactions may run for an extended period, such unresolved RTS can create conflicts for a non-negligible number of writers. Unfortunately, this behavior is difficult to detect or punish directly, as it is from a correct but slow client.

To mitigate this risk, RTS may be implemented as *leases*: each RTS is associated with an expiration time, after which it is ignored by writers. Expiration policies can vary across replicas and be adapted to individual clients. Clients that repeatedly allow their RTS to expire without completing their transactions are, from the system's perspective, functionally equivalent to malicious actors and can be deprioritized accordingly.

Read Only Transactions We briefly discuss an optional optimization for read-only transactions—transactions that perform no writes. By default, Basil assigns each transaction a fixed timestamp ts for which serves as its logical serialization point: the transaction is treated as if it occured atomically at time ts.

However, in many applications, it suffices for read-only transactions to observe *some* consistent snapshot, rather than a specific timestamp. Serializing reads at the pre-assigned ts may unnecessarily enlarge the transaction's conflict window. For instance, if the read-only transaction's assigned timestamp ts is greater than the largest version read v, then any concurrent writer with a timestamp between v and ts introduces a conflict, forcing either the reading or writing transaction to abort. To reduce such conflicts and improve commit success—both for the read-only transaction and concurrent writers—a client may dynamically adjust its timestamp downward to $min(ts, v)$ before entering validation. This constrains the conflict window to only include transactions that may have influenced the actual data read.

This optimization is not universally appropriate. While Basil does not enforce *strict* serializability,[7] some applications may still rely on local causality—for instance, requiring that reads reflect the effects of a client's own prior writes. Dynamically lowering a transaction's timestamp may break this expectation by introducing non-monotonic behavior: a client may inadvertently "time travel" and observe a system state that predates one it already influenced.

3.7.4 Timestamp Selection Alternatives

Basil optimistically allows clients to select their own transaction timestamps. To limit Byzantine abuse—particularly from clients that select artificially large timestamps to provoke read-write conflicts—Basil rejects reads and transactions with timestamps greater than $R_{Time} + \delta$, where R_{Time} is the replica's local clock and δ accounts for clock-skew and network latency. The value of δ can be client-specific, adjusted based on observed latency and behavior (*e.g.*, made smaller for clients that frequently overshoot).

We briefly discuss two alternative mechanisms to mitigate the impact of high client-issued timestamps.

Scheduling at Replica local time A simple mitigation is for replicas to defer processing requests until their timestamps are no longer in the future—*i.e.*, until the timestamp matches or precedes the replica's local clock. This prevents premature validation of transactions with overly high timestamps, limiting their ability to interfere with concurrent operations while increasing their own conflict window. However, such deferral also introduces additional for correct clients that conservatively overestimate clock skew or

[7]Basil provides (Byz-) strict serializability if clocks are synchronized and timestamps reflect real time; otherwise, it guarantees Byz-serializability. As with Byz-serializability, external consistency only holds for correct clients—Byzantine clients may arbitrarily backdate timestamps to subvert real-time ordering.

network delay, potentially increasing their risk of abort.

Sourcing Bounded Timestamps Rather than selecting timestamps freely, clients may obtain a bounded timestamp by querying at least $2f + 1$ replicas (within one shard) and choosing the $f + 1$st largest response. This value is both upper and lower bounded by correct replicas, making it robust to Byzantine manipulation The main drawback is the additional round-trip latency and the need to attach evidence proving the timestamp's legitimacy. To mitigate the latency overhead, clients may combine this mechanism with Basil's default optimistic timestamping. For example, a client can provisionally pick a timestamp and concurrently request a bounded one during execution. Prior to commit, it updates the transaction timestamp to the minimum of the two; if the adjusted timestamp falls below any read version in the ReadSet, the transaction must abort.

3.7.5 Streamlining Decision Logging

To confirm that a decision has been durably logged, a client must collect $n - f = 4f + 1$ matching decision confirmations (ST2R). In the common case—where the network is synchronous and there are no competing interested clients—a correct client issuing a ST2 request can *expect* to receive such a quorum. If the first $n - f$ replies do not match, the client can safely assume that some are Byzantine and continue waiting; all $n - f$ correct replicas will eventually respond with consistent decisions.

This guarantee breaks down if other clients concurrently attempt to log conflicting decisions. In such cases, the fallback protocol may be needed to reconcile divergent decisions. To avoid unnecessary fallbacks, replicas typically allow a short grace period during which only the originating client is expected to submit a decision. This optimization is best-effort and may fail under high tail latencies or asynchronous networks.

If a client fails to gather $4f + 1$ matching ST2R decisions in a timely manner, it must invoke the fallback protocol. However, it may still wait in parallel for additional matching replies from the same view. This is safe as the fallback process must preserve any decision that *could have* returned.

To streamline progress when replicas *nearly* match—*e.g.*, when only one or a few replies deviate—we introduce an optional third logging phase (Stage 3). A client that receives $3f + 1$ matching ST2R messages may broadcast a Stage 3 message ST3 $:= \langle id_T, view, decision, \{\text{ST2R}\}\rangle$ containing evidence of the $3f + 1$ matching ST2R responses. This allows a client to make progress even if the first $n - f$ requests include up to f Byzantine participants that lie about their decision. Replicas that receive a valid ST3 message log the decision and respond with a ST3R $:= \langle id_T, view, decision\rangle$ message. The client considers the decision durably logged once it has received $3f + 1$ matching ST3R responses.

Stage 3 is purely auxillary, and does not interfere with Stage 2: clients may proceed to Writeback upon receiving either $4f + 1$ ST2R's or $3f + 1$ matching ST3R's. The latter path may be faster when the last few replicas suffer from high tail latency.

Supporting Stage 3 requires a small change to the fallback reconciliation protocol. Replicas now include their ST3 decision in ELECTFB messages, which carry both the ST2R and ST3R decision. In case of a conflict within the same view, ST3R decisions take precedence. Specifically, $f+1$ matching ST3R's suffice to trigger recovery. This is safe because the existence of $f + 1$ ST3R decisions implies that at least one correct replica observed $3f + 1$ matching ST2R messages. By quorum intersection, no two valid but conflicting ST3 messages can exist within the same view. Likewise, no conflicting ST3 decision and a successfully logged ST2 decision can exist in the same view. Therefore, any valid ST3 provides conclusive evidence that a decision was logged. When conflicting recoverable decisions exist across views, the one from the highest view takes precedence.

Trading off Commit and Abort Latency Basil can further harden commit latency by asymetrically trading it off against abort latency (or vice versa): commit decisions can finalize with just $3f + 1$ matching ST2R replies, as long as abort decisions are required to complete in Stage 3 (*i.e.*, they cannot finalize in Stage 2). This modification strenghtens the latency bound for commits, while preserving safety: if a ST3 abort decision exists, it is impossible for $3f + 1$ conflicting ST2R commit decisions to exist in the same view (and vice versa).

To support this asymmetry, the fallback reconciliation rules must be adjusted: $f + 1$ ST2R commit votes suffice to recover (and repropose) a commit decision in Stage 2, while $f + 1$ ST3R abort votes suffice to replay Stage3. Note that if a transaction committed (using $3f + 1$ matching ST2R replies) then every ElectFb quorum of size $4f + 1$ *will* recover the commit decision. The analogous holds true for abort decisions in Stage 3. Within the same view, a valid ST3 abort decision takes precedence over a valid ST2 commit decision. If there is no valid commit decision ($f + 1$ ST2R commit votes), but there are $f + 1$ ST2R abort votes, we replay Stage 2 with abort as it is the only safe decision. As usual, decisions from higher views override those from lower views.

This optimization is well-suited for applications where aborts are expected to be rare. In such cases, the common (commit) path completes in two round-trips, while the less frequent aborts pay the cost of Stage 3. Notably, aborts can still complete early via the Stage 1 abort fast path with just $3f + 1$ matching ST1R abort votes.

3.7.6 Reducing Deployment Cost

Separating Validation and Storage By default, Basil uses a total of $5f + 1$ replicas. Although all replicas must participate in concurrency control (CC) to mainain Byz-serializability and Byzantine independence, not all replicas need store the full dataset. For servicing reads, it suffices for any given key to be stored at $2f + 1$ replicas. The remaining replicas need only store metadata—such as keys and version information—for prepared and committed transactions; values themselves are not required to perform the MVTSO validation check. This separation enables more flexible deployments.

For example, Basil can operate with $2f + 1$ storage replicas and $3f$ additional "witness" replicas that participate in CC but do not store full data. Alternatively, for scalability, the storage and CC roles can be assigned to physically distinc nodes—*e.g.*, $5f + 1$ lightweight CC witnesses combined with $2f + 1$ dedicated storage nodes. Finally, storange responsibilities may be distributed across replicas by partitioning keys, allowing different subsets of $2f + 1$ replicas to store disjoint key ranges.

Basil with $3f + 1$ Replicas

We briefly discuss how Basil can, in theory, be instantiated with only $3f + 1$ total replicas—referred to as Basil3. While this configuration reduces the total number of replicas, it is not recommended in practice: it sacrifices Byzantine independence, weakening the systems' robustness against Byzantine influence, and incurs an extra commit round. Nevertheless, Basil3 may be of pedagogical interest and applicable in resource-constrained settings where the deployment cost of $5f + 1$ replicas is prohibitive.

No Byzantine Independence To ensure both safety and liveness with $3f + 1$ replicas, the CommitQuorum (CQ) size must be fixed at $2f + 1$: smaller quorums lack quorum intersection for conflicting transactions, while larger ones exceed the number of replies a correct client can expect (*i.e.*, $n - f = 2f + 1$, from correct replicas).

As a result, if a client fails to collect $2f + 1$ commit votes, it must pessimistically abort the transaction—even if only a single Byzantine replica voted to abort—breaking Byzantine independence. While clients may attempt to wait for additionally replies, this cannot be done reliably under partial synchrony, where remaining correct replicas may be arbitrarily delayed.

Worse, Byzantine clients can sabotage their own transactions by colluding with faulty replicas to force aborts, making their prepared writes unreliable. To remain live, correct clients must then avoid depending on such writes—limiting concurrency and degrading performance.

No Commit Fast Path Basil3 cannot safely support a single-round commit fast path: even if all $3f + 1$ replicas vote to commit, the decision is not durably logged. For example, a recovering fallback leader might observe only $f + 1$ commit votes—insufficient to conclude that a CommitQuorum (CQ) ever existed. As a result, single-round commit fast paths are unsafe in Basil3 and must be disabled. Abort fast paths, however, remain viable: receiving $2f+1$ abort votes rules out the existance of any CQ, ensuring the abort decision is durable.

Additional Logging Round Basil3, further, cannot reliably ensure durability of commit or abort decisions in a single logging round (*i.e.*, Stage 2 alone does not suffice). Consider a case where a client receives $n - f = 2f + 1$ matching ST2R commit replies—seemingly enough to return commit and proceed to Writeback—while the remaining f replicas locally log an abort. During recovery, the fallback leader must choose a decision based on any $n - f$ ELECTFB votes, of which up to f may be Byzantine. If those

f replicas flip their vote from commit to abort, the leader might observe only a single commit and $2f$ aborts—insufficient to safely conclude that the transaction indeed committed, breaking safety. Critically, this situation is indistinguishable from one in which the client observed $2f + 1$ ST2R abort votes and returned accordingly.

A third round—Stage 3—that reaffirms the ST2R decisions is required to durably log a decision. While the protocol structure mirrors the optional Stage 3 variant describe in Section 3.7.5, Basil3 treats Stage 3 as mandatory: a decision is considered durable only *after* successfully completing this round.

In Basil3, a decision is considered durably logged once $f + 1$ correct replicas have accepted ST3 $:= \langle id_T, view, decision, \{\text{ST2R}\}\rangle$ message. This message must include evidence of $2f + 1$ matching ST2R responsens, guaranteeing that no conflicting (valid) ST3 messages can exist within the same view. Upon receiving a valid ST3, replicas log the decision and persist the accompanying {ST2R} evidence to support recovery. A client may proceed to the Writeback phase after receiving $n - f = 2f + 1$ matching ST3R $:= \langle id_T, view, decision\rangle$ acknowledgments.

The fallback reconciliation rule is adjusted accordingly. Each replica casts its ELECTFB vote based on the latest stage it has reached (*e.g.*, Stage 2 or Stage 3) and includes the corresponding decision evidence—either the CQ or AQ that justifies a ST2 decision, or the set of ST2R messages that validates a ST3. Reconciliation gives precedence to Stage 3 decisions, and otherwise favors commit over abort. Since no two valid ST3 messages can exist in the same view, a single valid ST3 decision is sufficient to recover the outcome and re-initiate Stage 3 with the associated decision. If the ELECTFB quorum contains multiple ST3 messages from different views, the one from the highest view is selected. If no ST3 decision is available, but a valid ST2 commit decision is present, Basil3 attempts to replay Stage 2 with that commit evidence. Otherwise, it replays Stage 2 using a CommitQuorum (CQ) or AbortQuorum (AQ) of ST1 messages.

Notably, if Basil3 aborted during the Stage 1 abort fast path, no CQ can ever be formed—correct replicas never change their ST1 vote. Every ELECTFB quorum of size $n - f$ will contain at least one ST1R abort vote (a valid AQ), ensuring that the system proceeds to Stage 2 with an abort decision

An Optimization To reduce latency, Basil3 may reintroduce a fast path for Stage 2. The simplest approach is to treat a decision as durable once the client observes $n = 3f + 1$ matching ST2R messages. In this case, every ELECTFB quorum is guaranteed to contain at least $f + 1$ ST2 decisions (a majority)—or at least one corresponding ST3 decision—which suffices to safely replay Stage 2. If recovery reveals a valid Stage 2 decision from a higher view, it takes precedence of any Stage 3 decision.

An alternative, more easily satisfied fast path variant permits only on type of decision—*either* commit or abort—to conclude in Stage 2 with $2f + 1$ matching ST2R replies, but not both. This variant mirrors the earlier asymmetric optimization used to harden commit latency (§ 3.7.5). For example, commit decisions may be finalized in Stage 2, while abort decisions are required to complete Stage 3 (or vice versa). This assymetric rule preserves safety: if a ST3 decision exists, it is impossible for $2f + 1$ conflicting ST2R decisions to exist in the same view (and vice versa). Once again, the fallback reconciliation rules must be updated accordingly: a single valid ST3 abort decision takes precedene over a single valid ST2 commit decision in the same view—and, as usual, decisions from higher views override those from lower ones. In this configuration, the common (commit) path completes in two round-trips, while aborts pay the cost of Stage 3. Notably, aborts can still complete early via the Stage 1 abort fast path with just $2f + 1$ matching ST1R abort votes.

3.8 Related Work

Basil builds on a wide range of related work which we summarize briefly.

State machine replication (SMR)[160] maintains a total order of requests across replicas, both in the crash failure model [32, 93, 106–109, 112, 116, 121, 141, 142, 184] and in the Byzantine setting [12, 23, 27, 30, 34, 35, 53, 79, 80, 100, 101, 110, 120, 124, 126, 154, 190, 192], where it has served as a main building block of Blockchain systems [7, 8, 16, 27, 71, 98]. To maintain a total order abstraction, existing systems process all operations sequentially (for both agreement and execution), thus limiting scalability for commutative workloads. They are, in addition, primarily leader-based which introduces additional scalability bottlenecks [135, 166, 195] as well as fairness concerns. Rotating leaders [27, 35, 192] reduce fairness concerns, and multiple-leader based systems [12, 114, 135, 166] increase throughput. Recent work [85, 96, 97, 105, 197] discusses how to improve fairness in BFT leader-based systems with supplementary ordering layers and censorship resilience. Basil sidesteps these challenges by adopting a leaderless approach and tackles the broader impact of Byzantine actors—not just on ordering—through the stronger notion of Byzantine independence.

Fine-grained ordering Existing replicated systems in the crash-failure model leverage operation semantics to allow commutative operations to execute concurrently [102, 108, 115, 135, 137, 138, 148, 173, 188, 195]. This work is much rarer in the BFT context, with Byblos [17] and Zzyzyx [84] being the only BFT protocols that seek to leverage commutativity. However, unlike Basil, Byblos is limited to a static transaction model and introduces blocking between transactions that are *potentially* concurrent with other conflicting transactions; while Zzyzyx resorts to an SMR substrate protocol under contention. Other existing quorum-based systems naturally allow for non-conflicting operations to execute concurrently, but do not provide transactions [1, 41, 122, 125].

Sharding Some Blockchain systems rely on sharding to parallelize independent transactions, but continue to rely on a total-order primitive within shards [7, 98, 194]. As others in the crash-failure model have highlighted [138, 195, 196], this approach incurs redundant coordination and fails to fully leverage the available parallelism within a workload.

DAGs Other permissionless Blockchains use directed acyclic graphs rather than chains [152, 155, 157], but require dependencies and conflicts to be known prior to execution.

Byzantine Databases Basil argues that BFT systems and Blockchains are in fact simply databases and draws on prior work in BFT databases. HRDB [185] offers interactive transactions for a replicated database, but relies on a trusted coordination layer. Byzantium [66] designs a middleware system that utilizes PBFT [30] as atomic broadcast (AB) layer and provides Snapshot Isolation using a primary backup validation scheme. Augustus [146] leverages sharding for scalability in the mini-transaction model [6] and relies on AB to implement an optimistic locking based execution model. Callinicos [145] extends Augustus to support armored-transactions in a multi-round AB protocol that re-orders conflicts for robustness against contention. BFT-DUR [150] builds interactive transactions atop AB, but does not allow for sharding. Basil instead supports general transactions and sharding without a leader or the redundant coordination introduced by atomic broadcast.

Byzantine Clients Basil, being client-driven, must defend against Byzantine clients. It draws from prior work targeted at reducing the severity and frequency of client misbehavior [66, 122, 123, 145, 146, 150] and extends Liskov and Rodrigues' [122] definition of Byz-Linearizability to formalize the first safety and liveness properties for transactional BFT systems.

3.9 Conclusion

This chapter introduced Basil, the first leaderless BFT transactional key-value store supporting ACID transactions. Basil provides the abstraction of a totally ordered ledger while enabling highly concurrent transaction processing and guaranteeing *Byz-serializability*. Basil clients make progress independently, with *Byzantine independence* limiting the influence of Byzantine participants. During fault- and contention-free executions, Basil commits transactions in a single round-trip.

CHAPTER 4

PESTO: COOKING UP HIGH PERFORMANCE BFT QUERIES

The previous chapter introduced Basil, a novel client-driven approach to building Byzantine Fault Tolerant (BFT) datastores. Basil is a scalable, distributed BFT key-value store that supports parallel transaction execution without requiring total ordering. This design achieves high throughput and low latency, while mitigating the impact of Byzantine actors. However, Basil is limited to a key-value store interface and lacks support for higher-level computation primitives—such as queries—that are essential for many applications. To overcome this limitation, this chapter introduces Pesto, a scalable BFT Database that offers full SQL capabilities while maintaining high performance.

The context Decentralized applications that promise safe data sharing between mutually distrustful parties are being explored in sectors like finance [13, 15, 136], healthcare [5, 9], land records [103], secure key recovery [38], and general-purpose confidential computing [130, 131]. At their core lie Byzantine Fault Tolerant (BFT) consensus protocols [30, 73, 80, 101, 192], which provide a totally ordered, tamper-proof log distributed across mutually distrustful participants. This simple interface ensures that all parties observe the same set of operations, in the same order. In theory, the log can be materialized into a datastore consistent across all parties; in practice, however, accomplishing this is hard: applications must process the log, coordinate execution to ensure determinism, and handle potentially complex computations over the logged data.

The most common design for building a BFT datastore layers database functionality over BFT consensus [8, 55, 127, 151]. While conceptually simple, this approach is inefficient. Totally ordering all operations forgoes parallelism inherent to the workload. In theory, this overhead could be mitigated through sharding. Unfortunately, layering two-phase commit over consensus across many shards imposes significant coordination and cryptographic overhead [171, 195, 196].

Solutions that integrate consensus and database functionality have shown higher performance, but only for a basic key-value store (KVS) interface [150, 171]. Basil [171], the transactional and sharded BFT KVS presented in the previous chapter, eliminates the need for total ordering by efficiently integrating replication, optimistic concurrency control, and two-phase commit into a single, low-latency layer. Basil supports interactive transactions in a BFT setting, with performance competitive to crash fault tolerant (CFT) alternatives; however it only offers a limited KVS API and cannot easily (nor efficiently, § 4.7.2) express queries like joins, scans, or aggregations that are common to most real-world workloads.

In contrast, most centralized applications today look for the generality of databases. They expect support for (*i*) interactive transactions (transactions in which requests are interleaved with application code, which are preferred by developers over stored procedures [149]), (*ii*) a rich query language that supports query functionality (such as SQL), and (*iii*) horizontal scalability (the ability to safely partition data across shards).

Today, decentralizing these applications implies tolerating either limited SQL compatibility with low performance, or high performance but with a restricted KVS API. Truly general-purpose BFT databases should instead aim to *meet applications where they are*.

Towards expressive high performance BFT This chapter introduces Pesto, a general purpose BFT-database that achieves high performance while providing a powerful, expressive SQL query interface. Building on Basil's performant client-driven and ordering-free design, Pesto expands it to support full SQL functionality,[1] making it suitable as a drop-in replacement for many existing SQL databases.

[1]If life gives you Basil... make Pesto!

To achieve this, Pesto must overcome two challenges.

(*i*) *Maintaining serializability for arbitrary queries.* In key-value stores, ensuring the correctness of a query's results is straightforward as clients read each key individually. Commit certificates can assert the validity of read data tuples, and timestamps their recency; any further computation on read tuples is performed by the client itself, and thus inherently trustworthy. Pesto, instead, allows clients to submit complex queries that are executed server-side: asserting correctness thus requires trust in the computation performed by replicas.

A natural solution is *consistent* replication, which guarantees correctness by mandating that all replicas execute the same query on the same state, thus ensuring that the client receives enough matching responses to conclude that a correct replica vouched for the result. Unfortunately, this approach nullifies most of Basil's performance gains, as it requires all operations (not just queries) to be totally ordered at every replica.

(*ii*) *Extending concurrency control to generic queries.* Basil relies on an optimistic concurrency control protocol to maintain good performance in the common case while preventing malicious clients from stalling correct clients' transactions. Optimistic protocols, however, generally perform poorly for queries as they must, at least logically, lock the ranges of keys that may satisfy the query predicate; this results in high abort rates and low throughput.

Our secret sauce Pesto addresses both challenges with one key insight: when responding to a query, consistency *need only hold for the specific predicate of that query, not the entire database*. Pesto uses this observation to design a client-driven, snapshot protocol that ensures that, for a given query, replicas reply consistently. In many cases, results are already consistent, and no additional coordination is needed. When they are not, however, active resolution is necessary: Pesto clients dynamically establish a common snapshot of *relevant* state across replicas, thus ensuring reliably consistent results.

To improve concurrency, Pesto integrates query predicates and concurrency control (CC). Inspired by precision locks [92], it proposes a novel optimistic predicate-based CC check that only aborts concurrent transactions that violate query semantics. This allows Pesto to support high degrees of concurrency and eases the consistency requirement to only query *results*, and not the read state itself.

The benefits The results of our Pesto prototype are promising. On popular transactional workloads (TPC-C [182], AuctionMark [76], and Seats [76]) Pesto performs competitively with Peloton [77] and PostgreSQL [78], two unreplicated SQL databases, while reducing latency by factors of 2.7x (TPC-C) and more compared to classic layered designs (HotStuff [192]/BFT-Smart [176] + Peloton), and improving throughput by up to 2.3x (TPC-C). Microbenchmarks based on YCSB [39] further demonstrate that Pesto significantly improves the performance of analytical queries compared to Basil, and remains robust even in the presence of highly inconsistent or faulty replicas.

In summary, this chapter makes three contributions:

1. It presents a snapshot synchronization protocol that allows us to support arbitrary query computation atop inconsistent BFT replication (§ 4.4.5).
2. It introduces a novel semantics-based Optimistic Concurrency Control (CC) protocol which is carefully integrated with inconsistent replication and snapshot based execution (§ 4.5.1).
3. It presents and evaluates Pesto, a high performance distributed BFT DB that provides an interactive SQL transaction interface and can serve as a plug-and-play solution for existing SQL-based applications.

Roadmap The remainder of this chapter is organized as follows. Section 4.1 discusses existing approaches to building BFT databases and their shortcomings. Section 4.2 formalizes Pesto's correctness guarantees, and Section 4.3 outlines Pesto's architecture. We detail Pesto's query execution protocol in Section 4.4, and its concurrency control and commit protocol in Section 4.5. We discuss and prove Pesto's correctness in Section 4.6. Section 4.7 provides an evaluation of Pesto, Section 4.9 discusses related work, and Section 4.10 concludes.

4.1 Towards expressive, high speed BFT Queries

Existing applications are built on a well-established Database (DB) stack that provides high performance and enforces standard interfaces with rich query functionality. Developers are familiar with this stack and, we believe, would prefer to continue using it. In particular, they expect support for (*i*) transactional semantics, (*ii*) an expressive query interface, and (*iii*) built-in support for scaling workloads.

Transactions make it easy for developers to write bug-free code: they guarantee that sequences of actions will take effect atomically in the presence of concurrency or failures. Making transactions *interactive* (or *general*) allows for users to interleave transactional/database access code as part of their application directly, rather than using complex and cumbersome stored procedures, which are both hard to write and maintain [149]. SQL is the lingua franca today for data access and manipulation; it allows for expressive querying and powers decades of legacy code. Finally, most applications expect the ability to scale without having to reconfigure or rewrite the database. The database must consequently support partitioning the data into *shards* for horizontal scalability while safely supporting distributed transactions.

4.1.1 Layering Databases atop Consensus

The most straightforward way to implement a BFT DB is to employ a BFT consensus protocol (*e.g.*, PBFT [30]) to first totally order all operations, and then ingest the log into a DB engine of choice (*e.g.*, PostgreSQL [78]). Since all operations are ordered via consensus, the database at each replica will produce the same result. To execute (SQL) queries, clients simply submit them to the replicated backend (server-side execution) and wait for enough matching responses to confirm that at least one correct replica vouches for the result. This ensures (*i*) *data validity*: the query execution used a correct (valid) input state, *i.e.*, every value read corresponds to a committed write, (*ii*) *freshness* (bounded staleness): the query was computed using recent state, and (*iii*) *query integrity*: given the input state, the query was computed correctly according to its specification.

Unfortunately, this seemingly simple design performs poorly.[2] First, processing all requests sequentially, even those that don't conflict, is essential for ensuring consistency among the states of correct replicas. However, this approach eliminates the inherent parallelism of the workload. Sophisticated parallel execution engines [68] may recoup *some* of the lost performance, but are complex and cannot avoid establishing an initial total order. Second, interactive transactions may consist of several sequential requests, each requiring several round-trips of coordination to achieve agreement, resulting in high end-to-end latency. As a result, many existing systems [7, 8] limit transactions to single-shot stored procedures that are notoriously unpopular with developers [149]. Finally, scaling transactions horizontally, *e.g.*, via sharding, is inefficient, as layering two-phase commit (2PC) on top of internally replicated shards requires ordering each 2PC step. These fundamental limitations preclude practical deployments of this approach for OLTP systems.

[2]Though layered SMR-based designs appear modular and easy to compose, this simplicity is deceptive as a lot of complexity hides within the modules and their interplay.

4.1.2 Basil: An integrated BFT key-value store

To address these challenges, recent work proposes an innovative order-free approach to BFT-DBs. Basil [171], a serializable, distributed BFT key-value store (KVS), eliminates the need for totally ordering requests. Instead, it combines concurrency control (CC), replication, and two-phase commit (2PC) into a single, low-latency layer. In Basil, transactions are independently managed by clients and proceed in parallel whenever possible. Clients submit read operations (`GET` requests) to a subset of replicas and use their replies to identify fresh and valid responses. These replies include a Commit-Proof, which verifies the validity of the write that generated the returned value and its version. Writes (`PUT` requests) are buffered locally during transaction execution. To commit a transaction, clients initiate an efficient two-step commit protocol that simultaneously validates transaction execution results (to ensure serializability), computes a 2PC decision, and durably replicates the agreed upon result. When clients fail to complete transactions, Basil resorts to a cooperative recovery protocol that allows *any* client to recover and terminate incomplete transactions. This integrated database design has shown to be highly performant for a variety of popular OLTP workloads.

Basil, unfortunately, supports only `GET` operations: more complex, analytical queries must explicitly be re-structured. This is undesirable, as it increases the burden on application developers, and incurs coordination costs proportional to the size of a query's intermediate results. Consider a simple join query `SELECT * FROM tbl`$_x$`, tbl`$_y$ `WHERE` $x = y$ spanning two tables `tbl`$_x$ and `tbl`$_y$ (with primary keys x and y, respectively), both containing one million rows and overlapping in exactly one key. Executing this query requires first identifying the size of the tables (it is unknown to the client), and then issuing one million reads to `tbl`$_x$ and `tbl`$_y$ respectively; only then can the client determine locally the result, a *single* row.

4.1.3 Introducing Pesto

Pesto strives to retain Basil's performance and scalability while adding efficient support for complex SQL queries. This requires addressing two key challenges:

(*i*) Pesto must guarantee validity, freshness, and integrity for query results; it executes queries *server-side*, but requires that clients wait for enough matching replies to ensure that at least one correct replica vouched for the result.

Like Basil, Pesto prioritizes performance by not ordering requests. This approach carries a risk: even correct replicas may diverge during execution (*e.g.*, due to high contention) and produce different results. Pesto addresses this risk by introducing a synchronization protocol that dynamically, and only when needed, establishes common state snapshots (§ 4.4.5) for the current queries.

(*ii*) Pesto must ensure serializability for complex queries. Locking-based approaches are a non-starter in a Byzantine setting, as malicious clients can block progress by refusing to release their locks. Pesto must thus use optimistic concurrency control (OCC). Unmodified OCC, however, typically struggles with large data scans. Pesto addresses this issue by leveraging query semantics to create a novel semantics-aware OCC protocol that aborts transactions only if writes affect the result of concurrent queries, minimizing conflicts (§ 4.5).

4.2 Model

Pesto inherits the assumptions of Basil [171] and prior BFT work [30, 101, 192]. Pesto operates under partial synchrony [54]: it makes no timing assumption for safety, but for liveness depends on periods of synchrony.

Participants that adhere to the protocol are deemed *correct* while *faulty* (or *Byzantine*) participants may deviate arbitrarily. A strong but static adversary may coordinate the actions of faulty participants, but cannot break standard cryptographic primitives such as hashes, MACs, or digital signatures. We assume clients to be authenticated, and denote signed replica messages as $\langle m \rangle_\sigma$.

Pesto, for safety, enforces *Byz-serializability* [171], which, summarized curtly, ensures that all correct participants observe a sequence of states consistent with some sequential execution of the concurrent transactions. Byz-serializability on its own, however, does not ensure application progress; Byzantine actors could, for instance, still collude to systematically abort all transactions. We thus additionally enforce *Byzantine independence* [171]: in Pesto, no group of Byzantine participants may unilaterally decide the outcome of any operation, and therefore transaction progress is not subject to Byzantine abuse. To satisfy Byzantine independence, Pesto, like Basil, operates with $n = 5f + 1$ replicas, of which at most f may be faulty. Classic leader-based BFT protocols with a replication factor of $3f + 1$ [30, 101, 192], in contrast, cannot preserve Byzantine independence as a Byzantine client and leader may collude to front-run transactions or strategically generate conflicting requests

We place no bounds on the number of faulty clients. As is standard, Pesto cannot stop authenticated Byzantine clients from intentionally corrupting or deleting objects through legitimate transactions. Like Basil, Pesto does not verify whether authenticated clients adhere to the intended application semantics; instead it assumes that authenticated clients can be held accountable by an external service retroactively. In future work we aim to strengthen this property, and make access control fault tolerant as well

4.3 Pesto Overview

Pesto is a high performance distributed BFT DB that offers traditional SQL capabilities. It adopts the standard relational backend format of tables and rows, with rows uniquely identified by *primary keys*. Pesto supports standard SQL commands: `BEGIN`, read (`SELECT`, etc.), write (`INSERT, DELETE, UPDATE`, etc.), and `COMMIT` or `ABORT`. Transaction processing in Pesto, akin to Basil [171], follows the ethos of *independent operability*: execution is orchestrated by clients, and proceeds independently of all non-conflicting transactions. Replicas employ *inconsistent replication* [171, 195] and forgo totally ordering incoming requests; each replica may process client requests in any order, and in parallel. Transaction processing consists of two phases (Fig. 4.1).

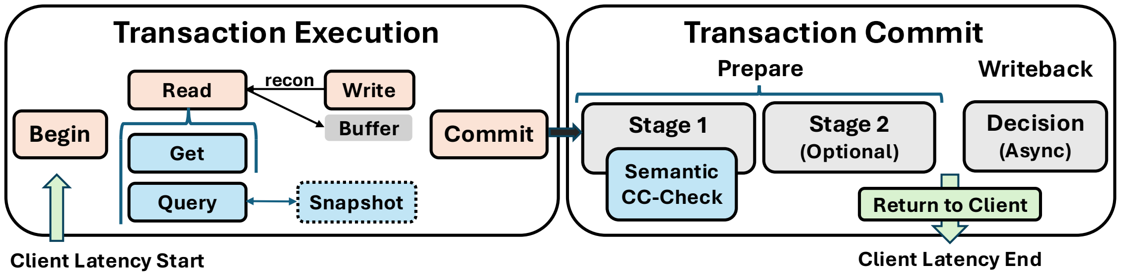

Figure 4.1: Pesto Transaction Processing Overview

Transaction Execution During execution, clients dynamically issue reads and writes. Write operations, which are often conditional, begin with an initial reconnaissance read to retrieve the rows to be modified. The client then modifies the target rows and buffers them until commit (§ 4.4.2). Simple reads that access only a single row can be processed via Basil's `GET` protocol (§ 4.4.4) and complete in a single round trip. All other queries (*e.g.*, more complex queries that access a variable amount of rows) proceed through Pesto's Range read protocol (§ 4.4.5). This protocol ensures that clients collect enough matching responses to assert that a correct replica confirms the result, thus ensuring validity, freshness and integrity. In many cases, replicas have the same relevant rows to produce matching results; when they do not, a client must first synchronize replicas on a common execution state via Pesto's snapshot protocol (§ 4.4.5). In the absence of failures, all (correct) replicas already eventually receive all data, and snapshots serve only to rendezvous; when Byzantine clients fail to fully disseminate their transactions, however, synchronization serves also as a recovery mechanism, ensuring that correct replicas exchange missing data.

Transaction Commit Transactions can commit if they do not violate (Byz-) serializability. To check for this, replicas locally compare concurrent transactions to determine whether a reader has missed a concurrent conflicting write, or vice versa (*prepare phase*). Crucially, Pesto leverages query semantics to determine which writes are potential conflicts: Pesto's SemanticCC (§ 4.5.1) considers concurrent operations conflicting only if a write affects a query's results.

Pesto, like Basil, opts to make writes optimistically visible upon successful validation (*prepared* writes); this reduces the opportunity for conflicts by up to two round-trips (the time to commit, discussed next), but requires carefully managing read dependencies to uphold (Byz-) serializability (§ 4.4.4, § 4.4.5).

Different replicas may validate conflicting transactions in different orders, leading to different votes. Two conflicting transactions, for instance, may both receive commit votes from different sets of replicas. To ensure safety, Pesto's Commit protocol (§ 4.5) requires clients to gather enough votes to guarantee that for any pair of conflicting transactions, at least one correct replica has validated both, resulting in the rejection of one transaction. Because transactions may involve multiple shards, Pesto aggregates vote tallies for each shard via Two-Phase Commit (Stage 1). For safety, this decision must be preserved across runs. In failure-free executions, transactions may complete in one round-trip, but an extra round-trip at a *single* shard is needed for durability if failures or network reordering arise (Stage 2). Finally, the client asynchronously notifies all replicas of the decision during an asynchronous *writeback phase*. If the decision is commit, a replica applies all buffered writes.

4.4 Transaction Execution

We first describe Pesto's transaction execution protocol. Since much of the system's complexity lies in the efficient handling of range queries, we devote the bulk of the section to this aspect.

4.4.1 Data structures

Pesto relies on two primary data structures, *transactions* and *versions*. Each version in Pesto corresponds to a unique write (insertion, update or deletion) and contains, in addition to column data, metadata necessary for maintaining serializability (Algorithm 2).

Algorithm 2 Row version

```
1: ts          ▷ timestamp $ts_T$ of writing transaction $T$
2: id          ▷ transaction identifier $id_T = h(T)$
3: status      ▷ enum:[commit, prepared]
4: col-vals    ▷ data values for each column; empty if deletion
```

Transactions receive a unique timestamp $ts_T := (localtime, ClientID, seq\text{-}no)$, selected client-side upon `BEGIN`. This timestamp implicitly establishes the transaction's final serialization order and allows Pesto to evaluate conflicts according to the designated ordering (§ 4.5.1). To avoid abuse by Byzantine clients that choose to fabricate large timestamps, replicas reject transactions that exceed their local clock R_{Time} by δ or more, where δ adjusts for client ping latency and clock skew. Pesto does not depend on δ for safety or liveness, though a well-chosen value can improve the system's throughput.

Algorithm 3 Transaction T meta-data.

```
1: ts                  ▷ transaction timestamp: (localtime, ClientID, seq-no)
2: ReadSet             ▷ (active) rows accessed: {(key, version)}
3: DepSet              ▷ read dependencies: {id_T'}
4: PredSet             ▷ read predicates: {pred} (§ 4.5.1)
5: WriteSet            ▷ buffered writes: {(key, col-vals)}
6: involved-shards     ▷ shards accessed: {shard-id}
7: id                  ▷ transaction identifier: Hash(T)
```

A transaction T additionally stores metadata documenting its execution (Algorithm 3). $ReadSet_T$ captures rows (and versions) accessed; $DepSet_T$ tracks read dependencies on visible but uncommitted versions (§ 4.4.2); $PredSet_T$ tracks query predicates (used for semantic concurrency control); and $WriteSet_T$ contains proposed row updates. Upon `COMMIT`, T receives a unique identifier id_T; it is computed as a cryptographic hash of T to prevent a Byzantine client from manipulating T's content—at worst, it can create a new transaction, which it can anyway always do. A client may also explicitly `ABORT` a transaction at any time before `COMMIT`, discarding all intermediate state without effect.

4.4.2 Serving Update Queries

We begin by describing how Pesto handles writes. As is standard in optimistic concurrency control, Pesto buffers writes locally until execution is complete. Doing so requires additional care when dealing with SQL statements that are typically *conditional*, and involve read-modify-write operations.

For example, an insertion only occurs if no row with the same primary key exists, while updates (`UPDATE` $table_x$ `SET` $x = x + 1$ `WHERE` $x = 5$) and deletes may depend on a predicate (*e.g.*, `DELETE FROM table`$_x$ `WHERE` x `=` 5). The rows to be written are thus only known *after* execution.

To handle conditional writes, Pesto splits write processing into two steps. First, a reconnaissance query fetches relevant rows Then, the client modifies or creates rows based on the original write statement. This approach allows Pesto to use the query interface to produce an intermediate query result and buffer new versions locally. For each row written, Pesto inserts a new write-entry into its current transaction's $WriteSet_T$. Pesto returns the number of rows written (possibly zero) to the application.

4.4.3 Servicing Reads

Reads in Pesto consist of SQL `SELECT` statements sent to replicas for execution. For efficiency, Pesto distinguishes between two types of reads, automatically deduced at runtime:

Point Reads read only a single row and explicitly identify the primary key of the table (akin to `GET` requests). Consider, for instance, a table U with columns a, b, and c, and composite primary key (a, b). The query `SELECT * FROM` U `WHERE` a `= 5 AND` b `= 'apple'` accesses the unique row with primary key (`5, apple`). Such reads can be efficiently executed by Basil's `GET` protocol (§ 4.4.4) and are guaranteed to complete in a single round-trip.

Range Reads (including scans, aggregate functions such as `Min`, `Max`, and joins), may instead scan through a variable (possibly unknown) number of rows. For efficiency, Pesto delegates the execution of complex queries to replicas and tries to collect $f + 1$ matching results to assert that at least one comes from a correct replica (which ensures validity, freshness, and integrity).

While simple, this approach does not guarantee liveness: because Pesto does not totally order operations at replicas, even correct replicas might not be consistent, and produce different results. At the time of reading, replica $R1$, for example, may have just inserted a new version with value $a = 10$ that is yet to be applied at $R2$; the two replicas will report a different number of rows for a query that reads a.

Waiting for inconsistencies to resolve and retrying the execution is not a reliable solution, as new concurrent writes can once again lead to mismatches. In fact, replicas might never be *fully* consistent. Unfortunately, forcing replicas to synchronize on all of their state is a non-starter, as that state can be large. Pesto instead uses a lightweight *snapshot synchronization* protocol that allows replicas to materialize *a consistent snapshot on demand, specific to a given query* (§ 4.4.5).

Once a read operation completes, the Pesto client return the result Q-RES to the application.

We describe the details of both read protocols next.

4.4.4 Point Read Protocol

To execute a point read, clients request valid versions from a quorum of replicas and select the freshest.

1: C → R: Client C sends read request to replicas.

C sends a read request POINT-READ := $\langle key, Q, ts_T \rangle$, containing the SQL query Q, the primary *key* it touches, and the transaction timestamp, to at least $2f + 1$ replicas.

2: R $\rightarrow$ C: Replicas process the client read and reply.

Replica R executes the submitted query Q and returns a result message POINT-RESP := $\langle$Q-RES, *Committed, Prepared*$\rangle_{\sigma_R}$. This response contains, respectively, the latest committed and prepared versions of the row identified by *key* with timestamps less than ts_T, if any exist. If these versions do not satisfy Q's predicate, R *still* returns a version, but indicates that the result Q-RES is empty. Tracking this version is necessary to ensure serializability during the commit stage.

Committed $\equiv$ (*version,* C-CERT) additionally includes a *commit certificate* C-CERT (§ 4.5.2) proving that *version* has committed, while *Prepared* $\equiv$ (*version,* $id_{T'}$) includes a digest identifier for the prepared transaction T' that wrote *version*.

3: C $\leftarrow$ R: Client C receives read replies.

C waits for at least $f+1$ replies to ensure that it receives at least one correct response and extracts the highest-timestamped version that is *valid*: a committed version must contain a valid C-CERT, while a prepared version (and result Q-RES) must be returned by at least $f + 1$ replicas. This guarantees (*i*) that C's transaction does not become dependent on fabricated versions, and (*ii*) that C returns a version no staler than if it had read from a single correct replica. Finally, C confirms Q's result Q-RES on the committed version itself, since replica-side computation is untrusted and Q may involve additional processing beyond simply reading the key—such as applying predicates or projections.

C returns Q-RES to the application and adds the selected (*key, version*) to its $ReadSet_T$. If *version* was only prepared, C additionally records a dependency $DepSet_T.insert(id_{T'})$ which will be used during T's Prepare phase to ensure (Byz-) serializability; T must not commit unless all the transactions in $DepSet_T$ commit first.

4.4.5 Range Read Protocol

Overview Processing arbitrary queries that may compute on ranges of rows requires additional care. To ensure validity, freshness, and integrity, a client must receive at least $f+1$ matching query results; this ensures that at least one correct replica vouches for the result. This is only reliable when correct replicas share the same state, which Pesto, by design, does not enforce for performance. Nonetheless, we observe that rows touched by a query often reflect state conistent across all correct replicas (§ 4.7). When they do not, Pesto creates its own luck by synchronizing replicas only on the rows accessed to agree on a common snapshot for the query. In fault-free cases, all (correct) replicas already eventually receive all data, and snapshots serve only to rendezvous; when Byzantine clients fail to fully disseminate their transactions, however, synchronization serves also as a recovery mechanism, ensuring that correct replicas exchange missing data.

Implementing the snapshot mechanism requires answering two questions: (*i*) What state should a snapshot contain and (*ii*) how to ensure that the snapshot proposal represents an up-to-date and valid state?

To compute a snapshot of a consistent state, Pesto uses the set of transaction *id*'s associated with the row versions that were read. This set uniquely identifies a specific state, and ensures atomicity (transactions are either included or not).[3] Individual row versions alone do not ensure atomicity, as Byzantine voters may selectively include versions. While this attack would be caught at commit time and thus not violate Byz-serializability, it would violate Byzantine independence!

Recording metadata for *every* row accessed during execution is often overly conservative. Most queries are predicated on a filter (*e.g.*, `name = 'Peter'`), and require

[3]Moreover, as transactions usually write to multiple keys, this set is typically smaller than the set of individual read versions.

only agreement on the set of rows relevant to the result (*i.e.*, those with name `'Peter'`, Figure 4.2). Pesto leverages this to include in its snapshots and read sets only the rows whose (latest versions) fulfill the query predicate (dubbed *active* rows). In most workloads, read predicates are highly selective and thus the set of active rows is usually small; this improves efficiency, and makes execution more likely to succeed as there are fewer opportunities for inconsistency.

Computation of active rows aligns naturally with index-based execution used in traditional SQL databases, which leverages predicates to reduce the number of rows that need be accessed (reducing query execution time by orders of magnitude) [164]. Pesto simply piggybacks on this strategy: since index search conditions are a subset of the query predicate, index scans will access all rows that affect the query result. We expand on the concept of active rows in § 4.5.1; they form the basis of Pesto's semantic concurrency control.

Protocol Details We next outline the details of the protocol. We do omit several pedantic details that impact our final implementation. Most readers will be happier skipping these details during their initial read; we defer rigorous correctness proofs, as well as additional discussion of details to Sections 4.6 and 4.8, respectively.

For simplicity, we assume that queries are satisfied by a single shard and that clients know the partitioning scheme. However, transactions may span multiple shards. Figure 4.2 illustrates an example execution. We briefly discuss distributed queries in Section 4.8.5.

1: C → R: Client C sends read request to replicas.

C sends a read request RANGE-READ $:= \langle Q := Query,\ ts_T \rangle$ to at least $3f + 1$ replicas (to ensure at least $2f + 1$ replies).

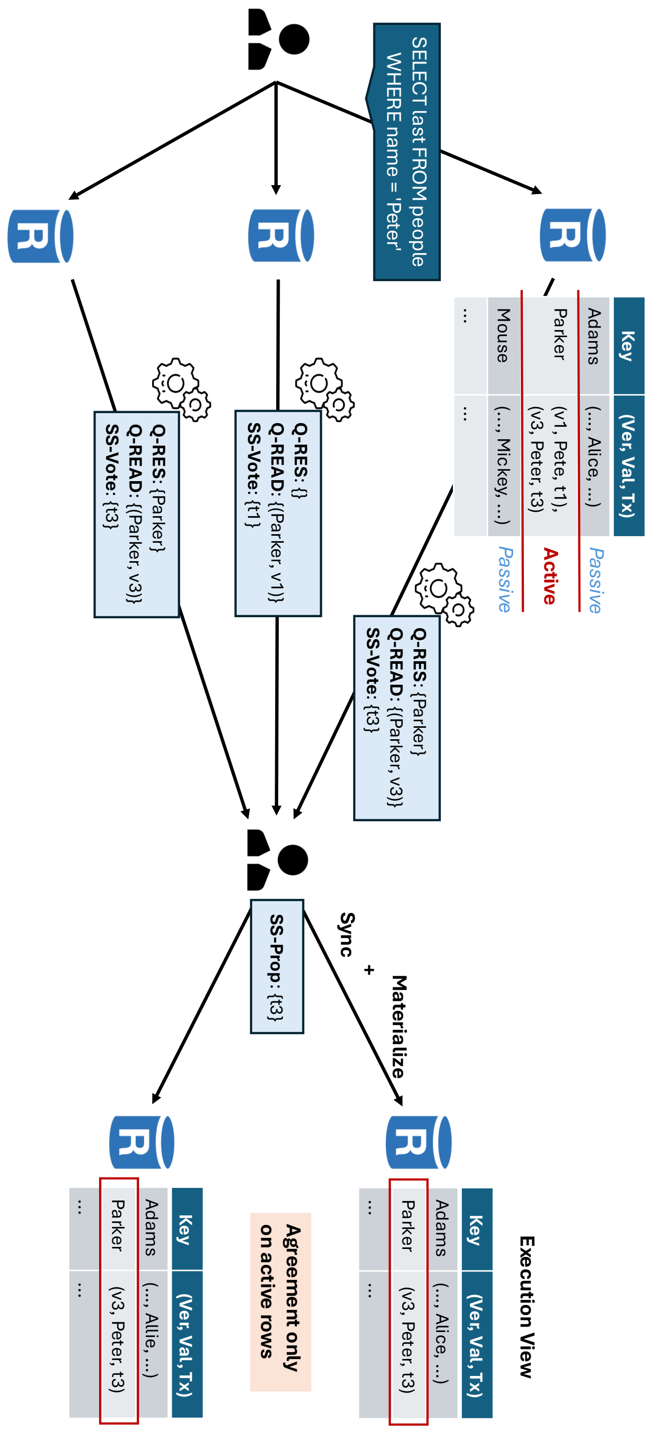

Figure 4.2: Life of a query (eager path disabled). The client broadcasts a query to a quorum of replicas who compute the query on their local state. Replicas return their query result (Q-RES)*, their (active) read set* (Q-READ)*—the rows that fulfill the query predicate—, and the associated snapshot vote* (SS-VOTE)*. The client aggregates a snapshot proposal* (SS-PROP) *and synchronizes replicas on a common execution state.*

2: R → C: Replicas process the client's read and reply.

A replica R executes the query Q on its local state (reading only versions no later than ts_T), and produces a query result Q-RES. For concurrency control purposes, R adds the active (key, $version$) pairs accessed during the computation (at most one version per key, the freshest version read) to a query read set Q-READ. In Figure 4.2, for instance, replicas record their latest version for the active key `'Parker'`. If a given version is only *prepared* (*i.e.*, tentatively committed), R additionally records a dependency on the version writer T', Q-DEP.$insert(id_{T'})$. Pesto's validation check uses this information to ensure that Q observes only serializable state.

Finally, R records as snapshot vote SS-VOTE the set of all transaction identifiers (id_{T*}) associated with the read set

R replies with RANGE-RESP-SS := $\langle$Q-RES, Q-READ, Q-DEP, SS-VOTE$\rangle\sigma_R$.

3: C ← R: Client C receives read replies.

Eager Path C waits for up to $2f + 1$ replies and tries to assemble $f + 1$ distinct replies with matching Q-RES and Q-READ, and *valid* dependencies Q-DEP (we defer discussion of the latter to a separate Section titled "Managing Dependencies"). This ensures that at least one correct replica vouches for the result and read set, which is necessary to correctly enforce serializability during validation.

If successful, C considers the read complete: it returns Q-RES to the application, and respectively adds Q-READ and Q-DEP to its ongoing transaction's $ReadSet_T$ and $DepSet_T$.

Snapshot Path If C cannot successfully complete a read, it enters the snapshot path. C tallies the snapshot votes and tries to propose a common execution state. To ensure liveness, a correct client must only propose to include transactions that exist, lest risk failing synchronization between replicas. Pesto must also ensure that faulty participants cannot cause a snapshot proposal to be artificially stale, as this would artificially extend the transaction's conflict window, making it much more likely to abort.

4: C → R: Client C proposes a snapshot to the replicas.

To generate a snapshot proposal, C selects all transaction ids present in $f + 1$ SS-VOTES and merges them into a proposal SS-PROP $:= \{(id_{T*}, \{\text{r}\})\}$, along with the ids of the replicas that suggested them. This filtering procedure ensures data validity: the set contains only transactions that at least one correct replica believes to be committed or prepared. In Figure 4.2, only $t3$ passes the filter.

Waiting to receive at least $2f + 1$ snapshot votes bounds staleness as it ensures that, if all correct replicas had this transaction in their state, this transaction will pass the filter and thus be included in the snapshot proposal ($f + 1$ proposals out of the $2f + 1$ necessarily come from correct replicas).

C then sends its SS-PROP to at least $3f + 1$ replicas.

5: R: Replicas process the snapshot and execute.

Upon receiving a snapshot proposal SS-PROP, a replica R checks whether it has already applied all included transactions: a transaction T' is considered applied once it has either (*i*) been explicitly aborted, or (*ii*) all of its write versions have been inserted into the respective rows. A replica might have received a transaction with $id_{T'} \in$ SS-PROP, but not yet applied it; query execution must wait for T' to be fully applied.

Synchronization If R has not yet applied a transaction T' in the snapshot proposal, it fetches it from its peers by sending a SYNC message containing $id_{T'}$ to the $f + 1$ replicas that included the T' in their SS-VOTE.

5.A: R → R: Replicas process a sync request and supply the transaction.

A correct replica ignores synchronization request for transactions it does not have. If a replica has T', it returns a message SUPPLY := (T', *status*, (C-CERT$_{T'}$/A-CERT$_{T'}$)), containing T', T''s current commit status, and, if T' has completed, its commit/abort-certificate. If C is correct and created its SS-PROP truthfully, then at least one voting replica is correct and will supply T'. If C is Byzantine and fabricated its SS-PROP, then synchronization will fail, affecting *only* the liveness of C's own query Q.

5.B: R ← R: Replicas process and apply a supplied transaction.

R processes SUPPLY messages according to the commit status, accepting and applying committed and aborted transactions after verifying the associated proof. If the decision is abort, R removes the transaction from any snapshot proposal it received; this may cause replicas to synchronize inconsistently (some replicas may not observe the abort) but is necessary as reading an aborted version will cause the query to abort too. Applying a transaction T' that is only prepared requires additional care. To uphold safety, R must not blindly prepare T': R might (due to inconsistency) detect a conflict that R' did not. To maximize consistency, however, Pesto opts to allow Q to read T' regardless of its validation outcome; after all, at least one correct replica part of SS-PROP considered T' prepared (or committed). Should T' fail validation, R applies T's write versions but makes them visible exclusively to Q.

Execution Once R has applied *all* transactions in SS-PROP it executes Q. During execution, a replica *tries* to read the freshest version associated with a transaction $id_{T'}$ included in SS-PROP (*i.e.*, $v3$ associated with $t3$ in Fig. 4.2). If the snapshot contains no version for a key accessed (this may happen, for instance, if a new relevant row is inserted after recording the SS-VOTE's) a replica simply reads the freshest commit version. In some cases, this is even preferable if the snapshot *does* have a version: if the freshest committed version is newer than the latest version in the snapshot, insisting on the snapshot may result in reading a stale version, ultimately causing the query's transaction T to abort. Reading the fresher committed version is thus often preferable. This may cause the client to not receive matching results; however, retrying only the query (and doing so early) is more cost-effective than continuing execution, and aborting the full transaction later.

R adds its chosen read version to Q-READ, and if the version has status *prepared*, it adds the versions' writer id_{T_w} to Q-DEP.

R returns a read reply RANGE-RESP := $\langle$Q-RES, Q-READ, Q-DEP$\rangle\sigma_R$.

6: C ← R: Client C receives read replies.

C considers a read successful upon receiving $f + 1$ replies with matching Q-RES and Q-READ, and valid Q-DEP (§ "Managing Dependencies"), as before. Because results can be legitimately inconsistent (*e.g.*, due to newer committed versions) C waits for up to $2f + 1$ replies to guarantee that at least $f + 1$ are correct (and thus do not fabricate inconsistency).

If C fails to receive matching replies, it restarts the snapshot path (by requesting a new set of SS-VOTES), and retries query execution.

Managing Dependencies

Pesto allows queries to read *prepared* versions but, for safety, must ensure that correct clients record dependencies. To maintain liveness, however, a client C must avoid including *fabricated* dependencies (or risk never completing its transaction). This raises a conundrum: C cannot afford to ignore legitimate dependencies, yet it should only accept dependencies that are vouched for by one correct replica. Unfortunately, while $f + 1$ replicas may agree on the read set Q-READ, their Q-DEP's might differ: some (correct) replicas may consider a candidate dependency $id_{T'}$ already committed and not include it in Q-DEP.

To determine whether to include a dependency C requires either (i) evidence that the dependency really exists, or (ii) evidence that a correct replica deems it already committed (and thus it need not be tracked). C can assert case (i) if a candidate dependency $id_{T'}$ appears across *any* $f + 1$ Q-DEP's or SS-VOTE's. Note that here, we do not require the result Q-RES nor read set Q-READ to match as C is only interested in gathering evidence for $id_{T'}$. If instead, C is unable to acquire evidence for $id_{T'}$, but gathers at least $2f + 1$ matching results it can conclude that at least one correct replica deems the dependency unnecessary because T' has already committed (case (ii)). If C can do neither, it cannot conclude legitimacy of the dependency, and must wait for additional replies (or retry the query).

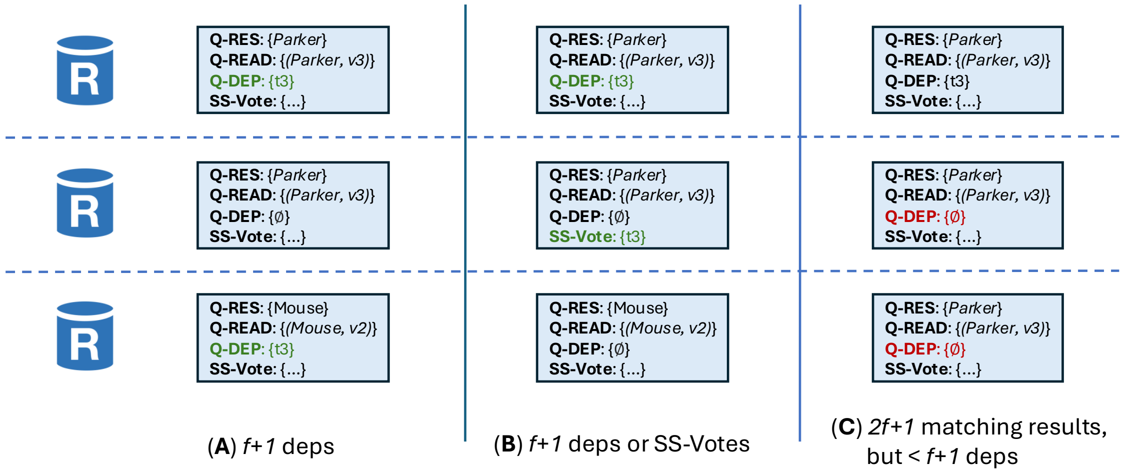

Figure 4.3: Illustration of the three criteria for determining dependency validity

An Example Figure 4.3 illustrates the three criteria. A dependency *dep* is valid only if it is vouched for by at least one correct replica. In the simplest case (not shown), there exist $f + 1$ replies that have matching query result (Q-RES), matching read set (Q-READ), *and* matching dependency set (Q-DEP). However, this may not be guaranteed, as some (correct) replicas may consider *dep* already committed and not include it in Q-DEP. To nonetheless determine the validity of *dep*, Pesto allows clients to check also the Q-DEP's and snapshot votes (SS-VOTE) reported by other replicas that do not have matching Q-RES or Q-READ.

In Figure 4.3, example **A**, for instance, the dependency $t3$ is only recorded in one of the first two matching replies (they have matching Q-RES and Q-READ); yet we require $f + 1 = 2$ dependency references to $t3$. $t3$ is, however, present in the third (inconsistent) reply, enough to conclude that $f + 1$ replicas deem $t3$ a valid dependency. Similarly, in example **B**, $t3$ is only recoreded in Q-DEP of a single replica, but it is also present in the SS-VOTE of another reply.

In some cases, dependencies may be valid but need not be recorded: if a client is certain that at least one correct replica deems the dependency unnecessary (for example because it has already committed the transaction), then we can ignore the dependency. In Figure 4.3, example **C**, for instance, all three replies match, but $f + 1$ replicas report no dependencies. Consequently, at least one correct replica deems *dep* unnecessary.

If a candidate *dep* cannot be found $f + 1$ times, nor safely considered unnecessary, then a correct client must consider the reply containing *dep* invalid and either wait for additional replies, or retry.

4.5 Transaction Commit

Once a transaction T completes execution, the client begins the commit process. Pesto adopts the core Basil [171] commit protocol, which we summarize for completeness

(§ 4.5.2). Unlike Basil, however, Pesto's concurrency control must efficiently and safely handle range queries; to this end, Pesto introduces a novel semantic based concurrency control, reminiscent of precision locking [92]. We first discuss how replicas locally perform validation. We then outline how clients aggregate individual replica votes to ensure (Byz-) serializability across replicas in a durable manner.

4.5.1 Concurrency Control (CC) Check

A replica votes to commit a transaction if the operations executed at that replica yields a serializable schedule. To check this, Pesto takes as starting point Basil's MVTSO algorithm. Each transaction is assigned a unique timestamp that predetermines its global serialization order. Transactions read the version with the highest timestamp still smaller than their own. In order to commit, no transactions may miss a write that they should have observed. As part of a validation phase, MVTSO checks transactions for pairwise conflicts: if a reading transaction T_R observed version v for key r, but a newer version v' ($< ts_{T_R}$) now exists, then T_R must abort.

This approach works well for point reads as only a single row is involved (and changes to the row's values likely affect the result returned). It does not, however, extend gracefully to *range* reads. Consider the transaction T in Figure 4.2 that issues a simple scan operation Q := `SELECT` *`last`* `FROM people WHERE name = 'Peter'` to a non-primary key `name`, and which returns as result only a single row (`'Parker'`).

For safety, T should record in its $ReadSet_T$ *all* rows present in table `people` as concurrent transactions might change the contents of any rows name column to `'Peter'`. In this particular example, T must additionally include a meta-data version on the table itself in order to prevent the concurrent insertion of new rows. Note that this example is not specific to Pesto's *range reads* but applies, of course, also queries explicitly written only using point reads.

To avoid Phantom Read anomalies [18], T must abort if even a single row in `people` is concurrently inserted, updated, or deleted; to commit, T must effectively acquire a (logical) lock on the *entire* table.

One can do better. A concurrent write that updates row `'Alice'` to `'Allie'` does not affect the result of Q, and thus does not violate serializability. Taking into account query *semantics* can significantly reduce the number of rows that need to be considered for range reads. Pesto leverages this idea to implement a semantics-aware CC check that determines whether concurrent writes affect the read predicate.

Read predicates

To implement semantisc-aware CC for queries, Pesto uses an approach common in databases. Each query (or sub-query) is broken into an operator tree with leaves consisting of full or partial table scans and an associated *filter predicate* (*e.g.*, `name = 'Peter'`). This allows Pesto to determine transaction conflicts by checking whether a write satisfies the filter predicates of a concurrent reader query (which may include multiple predicates). If yes, the write *could* be part of the read result. We find that filter predicates, in practice, account for the brunt of query selectivity, and only rarely unnecessarily abort writes that meet the filter criteria but do not change the end query result. Crucially, however, using filter predicates ensures that Pesto will never miss a write that does affect the query result.

Nested operations can, depending on the size of intermediary results, be represented either as coarse individual scans, or as multiple *instantiations*. Consider a simple query that joins two tables `SELECT * FROM tbl`$_x$`, tbl`$_y$ `WHERE x.name = 'Peter' AND x.id = y.account`, and, because `tbl`$_x$ has only a few rows with name `'Peter'` (one, in our example from Fig. 4.2), chooses to perform a Nested Loop Join [133]. Rather than deriving only two predicates Tbl_x : *x.name* = '*Peter*' and the very coarse Tbl_y : $True$, Pesto opts to *instantiate*, for each row r in the intermediary result of the sub-query (`SELECT * FROM tbl`$_x$ `WHERE x.name = 'Peter'`), a predicate Tbl_y : $x.id =< r.id >$.

Algorithm 4 summarizes the read predicate structure. We defer a discussion of *Table Versions* to Section "Making semantic CC efficient"; they serve only to improve the efficiency of transaction validation.

Algorithm 4 Read Predicate P.

```
1: table            ▷ The Table accessed
2: table_V          ▷ The Table Version at time of access
3: pred             ▷ The filter criteria, e.g., $x =? \wedge y > 5$
4: instantiations   ▷ Value instantiations, e.g., $x \in \{1, 7, 8\}$
```

Adjusting Range Reads We adjust the read replies (§ 4.4.5) sent by replicas to include the set of filter predicates Q-PRED associated with the query; replicas return a message $\langle$Q-RES, Q-READ, Q-DEP, Q-PRED$\rangle\sigma_R$ containing the query result, read set, dependency set, and set of predicates. Recall that replicas might have inconsistent state, and thus might instantiate different predicates for filters in nested queries: *e.g.*, the inner predicate of the query `SELECT names WHERE age = (SELECT age WHERE last = 'Parker')` depends on the `age` of `Peter Parker`. A client considers a read successful only if $f + 1$ replies have matching Q-PREDS. If so, it adds Q-PRED to its $PredSet_T$. This ensures that the recorded predicates are correct, and will safeguard serializabillity during the CC-check.

A simple, semantics-aware CC check

Given a read predicate P, Pesto distinguishes between the *active* read set (*ARS*)—all $(key, version)$ pairs that fulfill P (the *active* rows, stored in Q-READ)—and the *passive* read set—all other rows. Point reads, by design, only read active rows. Intuitively, the ARS captures all rows that are *relevant* to a read's computation (*i.e.*, contribute to the query result Q-RES). The passive read set is implicit, and need not be tracked.

To enforce Byz-serializability Pesto needs to ensure that the ARS is *fresh* and *complete*: (*i*) versions within the ARS are the most recent, and (*ii*) the ARS does not miss any relevant rows.

Algorithm 5 summarizes Pesto's CC-check; we defer formal safety proofs to Section 4.6. A replica R first performs some sanitization: it rejects transactions whose timestamps are too high (Line 1) or that claim possibly fabricated dependencies (Line 5). This ensures that Byzantine issued transactions do not disrupt progress of concurrent transactions. Replicas additionally reject transactions whose writes are *non-monotonic* (Line 3); we defer explanation to Section "Making semantic CC efficient". Next, a replica checks for serialization conflicts.

- **Read Conflicts** R first checks that T's ARS is *fresh* (Lines 7-9): there does not exist a write from a committed or prepared transaction T' that (*i*) is more recent than the version read by T and (*ii*) whose timestamp is smaller than ts_T (and thus should have been observed by T).

  R then checks that T's ARS is *complete*: for each predicate P, R determines if there exists a preceding write w from a transaction T' ($ts_{T'} < ts_T$) that (*i*) is *not* in T's ARS, (*ii*) is the freshest version visible to T, and (*iii*) fulfills P, and thus should have been in T's ARS (Lines 11-14). If w does not fulfill P, but T' is

Algorithm 5 SemanticCC-Check(T)

```
1:  if $ts_T > localClock + \delta$
2:      return Vote-Abort
3:  if $\neg isMonotonicWrite(T)$
4:      return Vote-Abort
5:  if $\exists$ invalid $d \in DepSet_T$
6:      return Vote-Abort
7:  for $\forall key, version \in ReadSet_T$
8:      if $version > ts_T$ return MisbehaviorProof
9:      if $\exists T' \in Committed \cup Prepared : key \in WriteSet_{T'}$
          $\wedge\ version < ts_{T'} < ts_T$
10:         return Vote-Abort, optional: $(T', T'$.C-CERT$)$
11: for $\forall P \in PredSet_T$
12:     if $\exists T' \in Committed \cup Prepared$ :
          $(P.table_v - grace) < ts'_T < ts_T\ \wedge$
          $\exists w \in WriteSet_{T'}$.
            $w.key \notin ReadSet_T \wedge \nexists w' : ts_{T'} < w'.TS < ts_T$ :
13:         if $P(w.col\text{-}vals)$
14:             return Vote-Abort, optional: $(T', T'$.C-CERT$)$
15:         if $\neg P(w.col\text{-}vals) \wedge riskyPrepared(T', w)$
16:             $DepSet_T$.insert($T'$)
17: for $\forall key, col\text{-}vals \in WriteSet_T$
18:     if $\exists T' \in Committed \cup Prepared.ts_T < ts_{T'}$:
          $ReadSet_{T'}[key].version < ts_T \vee$
          $(key \notin ReadSet_{T'} \wedge \exists P \in PredSet_{T'} : P(col\text{-}vals))$
19:         return Vote-Abort, optional: $(T', T'$.C-CERT$)$
20: $Prepared.add(T)$
21: 🕒 wait for all pending dependencies
22: if $\exists\ d \in DepSet_T : d.decision = Abort$
23:     $Prepared.remove(T)$
24:     return Vote-Abort
25: return Vote-Commit
```

only prepared, additional care is necessary: if T' were to abort and reveal (as next freshest write) a write w' ($ts_{w'} < ts_w$) that *does* fulfill P, then T may need to abort after all. In this case, R dynamically adds T' to $DepSet_T$ (Lines 15-16). We defer discussion of $P.table_v$ and *grace* to Section "Making semantic CC efficient".

- **Write Conflicts** Writes are checked analogously. R checks that writes of T do not cause reads of a *prepared or committed* transaction T' to miss a version (Lines 17-19): R checks that (*i*) the ARS of T' remains fresh, and that (*ii*) T's writes do not render the ARS of T' incomplete.

If R detects a direct conflict during validation, it immediately votes to abort T. Otherwise, if no conflicts are found, R *prepares* T and tentatively makes its writes visible to concurrent readers (Line 20). For safety, T may only commit if all of its read dependencies commit first (Lines 21-15). R therefore waits for these dependencies to resolve: it votes to commit T if all dependencies commit; otherwise it votes to abort T and rolls back T's tentative writes.

Making semantic CC efficient

Ensuring freshness for active reads is simple: it suffices to check for conflicts between the version read by T and ts_T. Ensuring completeness is less obvious: a newly arriving transaction T' that has a very old timestamp ($ts_{T'} << ts_T$) may still insert a new relevant row (or update the latest version of some relevant row), and thus must be validated for potential conflicts. To uphold safety, when a new transaction arrives, a replica R must therefore either (*i*) re-execute all of T's queries or (*ii*) check for conflicts against all transactions that ever wrote to the table; both of which are impractical (and can be exceedingly costly).

Re-introducing ordering We solve this problem by attaching, for each read predicate P of a transaction T, a concise summary of the (write) transactions that already happened prior to the read, and whose effects are thus included as part of the query result. Specifically, Pesto records a timestamp of the (then) latest write to the given table, denoted *table version* ($P.table_v$), and guarantees that all transactions with timestamp lower than $P.table_v$ are not conflicting (they are either part of the query result, or not relevant). As a consequence, R need only inspect the remaining transactions between $P.table_v$ and ts_T. Pesto enforces this invariant through *write monotonicity*: a new transaction T' may only write to a table if its timestamp is greater than any previously recorded *table version* (*i.e.*, is monotonic). Non-monotonic writers must be aborted (Alg. 5, Lines 3-4).

To implement this idea, we make two adjustments to Pesto's read protocol.

Hardening Range Reads First, Pesto must enforce monotonicity not only within one replica but across *all* replicas. Specifically, Pesto must ensure that the table version included part of a query reflects the latest committed transaction at any replica. By design, reads that miss fresher committed versions will be caught by Pesto's CC's freshness check. Table versions are different. They are the mechanism that ensures that current transactions are validated against *all* possibly concurrent conflicting transactions.

Our range read quorums (which require $f + 1$ matching replies, see § 4.4.5)—previously not required to intersect with transactions' commit quorums (§ 4.5)—must now be modified to intersect with these commit quorums in at least one correct replica. This requires clients to obtain table versions (and associated results) from at least $3f + 1$ replicas. This ensures that any relevant version with $TS < P.table_v$ is either (*i*) already applied, and thus observed by the query, or (*ii*) arrives only after updating the replica-local table version to $P.table_v$, and thus is aborted (it is non-monotonic).

While $3f + 1$ matching results are required for completeness, it is not strictly necessary to obtain $3f + 1$ matching *read sets*. As before, $f + 1$ matching read sets suffice to ensure that the selected read set is vouched for by a correct client (*i.e.*, it is not fabricated) and that it matches the query result. However, in practice, it is prudent to also check that all $3f+1$ replicas agree on the read set: although multiple read sets may yield the same result, stale ones may increase the likelihood of abort.

Table versions are only a coarse summary of the state, so two different (correct) replicas may return the same query result (Q-RES) and (active) read set (Q-READ) while reporting different table versions. To tolerate this, Pesto allows clients to complete range reads without matching table versions, and, for safety, simply selects the smallest table version as $P.table_v$. Byzantine clients may try to exploit this and report fabricated low versions. This does not affect safety, but makes CC-checks slower, as a larger range needs to be checked (even though it is not necessary for safety). To mitigate this, correct clients may discard replies with table versions that deviate significantly from the $f +$ 1st smallest reported version. Similarly, correct replicas may reject transactions whose reported predicate table versions are too low relative to the transaction's timestamp.

Accurately Representing Snapshots Second, Pesto must ensure that table versions accurately reflect the state read using snapshots. Consider a query that, in an attempt to read from a snapshot, reads a version several timestamps older than the freshest version of the row. For safety, Pesto must check for conflicts with concurrent transactions written *since* the read version, including, in particular, the versions that were skipped over (*e.g.*, a recent prepared version that was not included in the snapshot proposal SS-PROP). Replicas thus report as $P.table_v$ the minimum of the local table version and one less than the timestamp of the oldest encountered skipped version, ensuring it is included in the CC check. Concretely, $P.table_v = min(table\ version, min(\{ts_{skipped}\} - 1))$.

Relaxing Write Monotonicity Write monotonicity, in its simplest form, is overly harsh on writers: it does not account for varying transaction execution durations (recall, timestamps are selected at transaction begin), which may result in aborts for any "late" writers. To avoid this, Pesto relaxes the monotonicity requirement by adopting a sliding window approach. Replicas accept all transactions within the monotonicity threshold and a *grace* period, and accordingly validate a predicate P against writes between $P.table_v - grace$ and ts_T (Alg. 5, L. 12.1).

Grace periods offer a tradeoff. Larger grace periods offer writers more slack (reducing aborts), but may cause preparing readers to unnecessarily validate against transactions that were already part of their read state. Pesto attempts to soften this tension by distinguishing two grace tiers: (*i*) A first, short grace period reflects the assumption that most concurrent transactions arrive within close proximity, and requires validation against all transactions within the range. (*ii*) A second, larger grace period, captures (hopefully less frequent) longer or delayed transactions, and only validates against *non-monotonic* arrivals. Configuration of the grace periods does not affect safety, but if well chosen can improve performance.

4.5.2 Commit Coordination

Pesto adopts Basil's commit protocol, which was in detail in Chapter 3. We re-summarize it here for completeness.

Commit is entirely client-driven, and proceeds in two phases.

Prepare In the *prepare* phase the client submits its transaction T to all involved shards for validation. Replicas within a shard independently perform the local concurrency control (CC) check (Alg. 5), voting on whether committing T will violate Byz-serializability. Notably, replicas may process transactions in different orders, and thus even correct replicas may vote differently. The prepare phase ensures mutual exclusion, that no two conflicting transactions may both commit. To this end, the client tallies the replica votes of each involved shard into a single *shard-vote*. A transaction is deemed committable only if enough replicas vote to commit such that no conflicting transaction will ever also become committable. For transactions that access multiple shards, the client additionally aggregates shard-votes as part of a two-phase commit (2PC) protocol: T commits if all shards vote to commit, and aborts otherwise.

The *prepare* phase consists of two sub-stages. In stage ST1, the client collects, for each shard that T accesses, *commit or abort votes* from all of the shard's replicas. These votes are then used to make the 2PC decision. If the client receives sufficiently many votes to conclude that the 2PC decision will remain durable across failures, it proceeds immediately to the *writeback* phase.

A shard-vote is considered durable iff it can be independently retrieved by any client (*i.e.*, any vote tally quorum produces the same decision). Durable shard votes form a *vote certificate* V-CERT $:= \langle id_T,\ S,\ Vote,\ \{\text{ST1R}\}\rangle$; we dub shards with V-CERT *fast*, and shards without *slow*. Shard-votes are tallied as follows:

1. **Commit Slow Path** ($3f + 1 \leq$ commit votes $< 5f + 1$): The client has received at least a *CommitQuorum* (CQ) of votes, where $|CQ| = \frac{n+f+1}{2} = 3f + 1$ *Vote-Commit*.
2. **Abort Slow Path** ($f + 1 \leq$ abort votes $< 3f + 1$): A collection of $f + 1$ abort votes constitutes the minimum *AbortQuorum (AQ)* that preserves Byzantine independence. Pesto clients are guaranteed to observe (at least) either a *CQ* or an *AQ* (of size $3f + 1$ or $f + 1$ respectively).
3. **Commit Fast Path** ($5f + 1$ commit votes): No replica reports a conflict, and thus any

possible quorum of size $n - f$ will contain sufficiently many commit votes to form a CQ.

4. **Abort Fast Path** ($3f + 1 \leq$ abort votes): T conflicts with a prepared transaction, and no other quorum can receive sufficiently many commit votes to form a CQ.
5. **Abort Fast Path** (One abort vote with a C-CERT for a conflicting transaction T'). T (provably) conflicts with a committed transaction; any quorum will conclude abort.

C decides to commit T if all shards vote to commit, and otherwise aborts T. If *all* shards voting to commit (or analogously if a single shard voting to abort) are fast (and thus the votes are durable), C aggregates the respective V-CERT's and proceeds immediately to the *writeback* phase. If a single committing shard is slow (or the one aborting shard is slow) C must first complete Stage ST2. Rather than make the *votes* of slow shards durable, Pesto opts to replicate the tentative 2PC *decision*. To do so, C selects *one* of the involved shards, henceforth denoted as S_{log}, and logs on it the decision; S_{log} is chosen deterministically depending on T's id. C records a single durable V-CERT$_{S_{log}}$:= $\langle id_T,\ S,\ decision,\ \{$ST2R$\}\rangle$.

In the absence of failures and contention, Pesto's *fast-path* allows clients to commit a transaction in a single round-trip; otherwise, one additional round-trip is required.

Writeback Once the decision is durable, C notifies its application of T's outcome, aggregates shard votes into a decision certificate C-CERT/A-CERT := $\langle id_T,\ decision,\ \{$V-CERT$_S\}\rangle$ (commit or abort, respectively), and asynchronously informs all involved shards. On the fast path, C-CERT consists of the commit V-CERT's from all involved shards, while an A-CERT need only contain one shard's abort V-CERT. On the slow path, both C-CERT/A-CERT simply include V-CERT$_{S_{log}}$. Replicas that commit T create new version for each written row. Additionally, replicas notify pending dependencies (transactions that read only prepared values of T) on the outcome of T.

Recovery In case of client failures, a cooperative Fallback protocol allows other clients to terminate ongoing transactions. We defer details of recovery to Basil, presented in Chapter 3; they do not affect how Pesto processes queries.

4.6 Correctness

We show that Pesto upholds Byz-serializability and Byzantine independence.

Byz-serializability captures Pesto's *safety* requirement: it ensures that all correct participants observe a serializable state. While Byzantine participants may choose to view a non-serializable state, they cannot compromise the safety of correct participants.

We note that *liveness*, in a traditional sense, does not apply to general-purpose interactive transactions. Whether a transaction commits depends on runtime contention and concurrency—factors outside the protocol's control. Transactions facing contention may need to abort and retry; avoiding aborts with *certainty* requires a-priori knowledge of a transaction's read and write sets, which is generally unavailable for interactive transaction workloads. Nonetheless, Pesto is designed to make *as much progress as possible*. In particular, transaction progress should be decoupled from Byzantine behavior. Byzantine independence formalizes this requirement: no operation—especially transaction outcomes—should be unilaterally determined by Byzantine participants.

Additionally, Pesto should ensure progress in the *absence* of contention. Specifically, if no new contending transactions arrive concurrently, all ongoing (correct clients') transactions should eventually commit. To model this scenario, we assume a contention-free time t_{CF} after which no further conflicting transactions are submitted. We show that under this condition, Pesto guarantees transaction commit after t_{CF}.

4.6.1 Definitions

For completeness, we restate the formal definitions of Byz-serializability and Byzantine independence introduced in Chapter 3.

A transaction T contains a sequence of read and write operations terminating with a commit or an abort. A history H is a partial order of operations representing the interleaving of concurrently executing transactions, such that all conflicting operations are ordered with respect to one another. A history satisfies an isolation level I if the set of operation interleavings in H is allowed by I.

Additionally, let C be the set of all clients in the system; $Crct \subseteq C$ be the set of all correct clients; and $Byz \subseteq C$ be the set of all Byzantine clients. A projection $H|_{\mathscr{C}}$ is the subset of the partial order of operations in H that were issued by the set of clients $\mathscr{C}$.

Definition (Legitimate History) *History H is legitimate if it was generated by correct participants, i.e.,* $H = H_{Crct}$.

Definition (Correct-View Equivalent) *History H is correct-view equivalent to a history H′ if all operation results, commit decisions, and final object values in* $H|_{Crct}$ *match those in H′.*

Definition (Byz-I) *Given an isolation level I, a history H is Byz-I if there exists a legitimate history H′ such that H is correct-view equivalent to H′ and H′ satisfies I.*

Pesto specifically guarantees Byz-serializability.

Definition (Byzantine Independence) *For every operation o issued by a correct client c, no group of participants containing solely Byzantine actors can unilaterally dictate the result of o.*

Notation In the following we refer to the unique identifier of a row as the *row-key*. In practice, this is a unique encoding of the rows primary key. For each row-key, there may exist multiple *row-versions*, each corresponding to the write of a unique transaction.

4.6.2 Correctness Sketch

We adopt and extend Basil's proof of Byz-serializability to Pesto. It proceeds in four steps:

First, we prove that Pesto's concurrency control ensures that each correct replica generates a locally serializable schedule. We adopt Adya's formalism here [4]: an execution of Pesto produces a direct serialization graph (DSG) whose vertices are committed transactions, denoted T_t, where t is the unique timestamp identifier. Edges in the DSG are one of three types:

- $T_i \xrightarrow{ww} T_j$ if T_i writes the version of object x that precedes T_j in the version order.
- $T_i \xrightarrow{wr} T_j$ if T_i writes the version of object x that T_j reads.
- $T_i \xrightarrow{rw} T_j$ if T_i reads the version of object x that precedes T_j's write.

We assume, as does Adya [4], that if an edge exists between T_i and T_j, then $T_i \neq T_j$. An execution is serializable if the DSG is cycle-free. To prove Lemma 9 it suffices to prove that if there exists an edge $T_i \xrightarrow{rw/wr/ww} T_j$, then $i < j$ (*i.e.*, the timestamp of the outbound vertex is smaller than the timestamp of the inbound vertex: $ts_{out} < ts_{in}$).

Based on this, we define a notion of conflicting transactions: informally, a transaction T_i conflicts with T_j if adding T_j to a history containing T_i would cause the execution to violate Byz-serializability.

We show:

Lemma 9 *On each correct replica, the set of transactions for which the CC-Check returns* `Vote-Commit` *forms an acyclic serialization graph.*

Next, we must show that Pesto's commit protocol ensures that decisions for transactions are unique.

Lemma 10 *There cannot exist both an* C-CERT *and a* A-CERT *for a given transaction.*

Likewise, the commit protocol must ensure that no two conflicting transactions may both commit.

Lemma 11 *If T_i has issued a* C-CERT *and T_j conflicts with T_i, then T_j cannot issue a* C-CERT.

It follows that Pesto satisfies Byz-serializability.

Theorem 7 *Pesto maintains Byz-serializability*

Since Pesto adopts the core Basil commit protocol, the proofs of Lemmas 10 and 11 follow directly from Basil; we refer to Section 3.5 for the full proofs. We prove that Pesto's concurrency control upholds Lemma 9.

Likewise, it follows directly from Basil that Pesto's point read and commit protocol are Byzantine independent. We prove additionally that Pesto's range read protocol upholds Byzantine independence, and ensures progress in absence of contention.

Theorem 8 *Pesto's range read protocol is Byzantine independent.*

Theorem 9 *Pesto's range read protocol guarantees successful termination after t_{CF}.*

4.6.3 Byz-Serializability

We first show that Pesto's range read protocol guarantees validity and integrity for correct clients. This ensures that, given a serializable input state, any query issued by a correct client yields a serializable result. Furthermore, the returned *ReadSet*, *PredSet*, and *DepSet* accurately reflect the result and preserve serializability. We do not assess the correctness of reads performed by Byzantine clients; under Byz-serializability, such clients are responsible for their own consistency.

Lemma 12 *Successful range reads issued by correct clients uphold data validity and query integrity, and produce correct concurrency control meta-data.*

Proof. Range read execution succeeds if a (correct) client receives $3f+1$ matching read results, read sets (and valid dependency sets), and predicate sets. It follows that at least one correct replica vouches for the result and asserts that it corresponds to the reported read and predicate sets. A correct replica will only read valid row-versions, and will perform the query computation truthfully (*i.e.*, with integrity). Finally, if at least one correct replica reports a row-version as only tentatively committed (prepared), a correct client will register a dependency, or fail the range read. □

Notably, any details relating to snapshot synchronization do not impact data validity and query integrity. They affect only responsiveness, Byzantine independence, and freshness.

For completeness, we show correctness also for point reads. We note that point reads, by design, do not rely on server-side computation. A point read may perform simple data transformations on a row-version; however, these may be performed client-side—integrity is thus a given.

Lemma 13 *Successful point reads issued by correct clients uphold data validity and produce correct concurrency control meta-data.*

Proof. Point reads return either a committed or prepared row-version. Committed row-versions are supported by a Commit-Proof which, by definition, proves the validity of the write. Clients can execute their query on the associated row-value to confirm the result. Prepared row-versions, in turn, are only selected by correct clients if backed by $f + 1$ replicas, thus asserting that at least one correct replica has tentatively prepared the value. Correct clients include the (unique) row-key and row-version in their read set, as well as a dependency if the read row-version was prepared. □

Transaction Conflicts In the following, we refer to an execution of Pesto as the set of committed transactions. An execution of Pesto is (Byz-) serializable if the execution results of all (correct clients') transactions are equivalent to *some* serial ordering of all committed transactions. Pesto simplifies this objective by making the serialization order explicit: transactions in Pesto are assigned a position in the serialization order via their timestamp. A Pesto execution consequently upholds (Byz-) serializability if the execution results of all committed transaction is consistent with the timestamp-induced serialization order.

We say that a pair of transactions T_i, T_j *conflicts* if $ts_i < ts_j$, yet T_j's execution results are not compliant with the serial order. By design, T_i and T_j may only conflict if T_i produces a write that T_j should observe, *i.e.*, a write that changes the result of T_j's read. This corresponds to a *rw*-edge in the DSG where $T_j \xrightarrow{rw} T_i$, thus violating our objective that $ts_{out} < ts_{in}$ for all edges in the DSG. All other cases (both transactions read, both transactions write, or T_i reads) are conflict-free: writes are applied to a multi-version store and indexed by their timestamp, and reads of T_i exclusively read versions

$\leq ts_i$. It thus follows immediately that all *ww* and *wr* edges in the DSG uphold $ts_{out} < ts_{in}$ (we refer to Section 3.5 for full proof).

Definition (Transaction Conflict) *T_i and T_j conflict if T_i produces a write x_i to a row x read by T_j, but (i) T_j does not observe x_i, and (ii) there exists no other transaction T_k with $ts_i < ts_k < ts_j$ that writes x.*

We show that Pesto's concurrency control (CC) ensures that the set of transactions prepared by any given correct replica is conflict-free (or, in Adya's formalism, the DSG is cycle free). We re-state Lemma 9 adjusted for our conflict terminology.

Lemma 9. *On each correct replica, the set of transactions for which the CC-Check returns `Vote-Commit` is free of pair-wise conflicts.*

For every query, the active read set (ARS), by definition, contains all row-keys for which the query predicate evaluates to true; if there is no predicate, the ARS contains all row-keys (for the given table).[4] It follows from Lemmas 12 and 13 that correct clients report in their transactions correct ARS and predicates.

Let T_i and T_j be conflicting transactions such that T_i writes a row-key x, and T_j reads x. By the definition of a conflict, $ts_i < ts_j$ and T_i is the *last* writer to x preceding T_j in the serialization order, and T_j read a version of x with $ts_k < ts_i$.

By design, T_i's write set must contain the write x_i, as Pesto only writes keys in the write set.

We distinguish two cases: (*i*) r is in the active read set (ARS) of T_j, but is not fresh, and (*ii*) r is in the passive read set (PRS) of T_j, and thus T_j's ARS is incomplete.

[4]Note that point reads always contain a predicate strong enough to identify a singular row. Point reads are thus active by design.

We first show that the CC-check ensures that a replica R only votes to commit transactions whose ARS is conflict-free, *i.e.*, any concurrent write that would render a read row-version stale leads to an abort.

Lemma 14 *Pesto's CC-check detects stale ARS.*

Proof. There are two subcases: a replica R either executes the check for T_i before the check for T_j or vice versa. Note, that if T_i or T_j pass the CC-check at R but do not commit globally, then nothing need be shown as there is no conflict. We thus assume that the first transaction to be checked becomes committed; it follows that no correct replica that has prepared the transaction will ever change its local status to abort.

T_i **before** T_j**.** If T_i has passed the CC-check on replica R (and T_i ultimately commits) then T_i must either be in the *Prepared* or *Committed* set when R executes the check for T_j. When the check for T_j reaches Line 9 in Algorithm 5 the abort condition is satisfied for T_j because $r_j(x) = ts_k < ts_i < ts_j$.

T_j **before** T_i**.** If T_j has passed the CC-check (and T_j ultimately commits) then T_j must either be in the *Prepared* or *Committed* set when the check is executed for T_i. When the check for T_i reaches Line 18 in Algorithm 5 the abort condition is satisfied for T_i because $r_j(x) = ts_k < ts_i < ts_j$.

It follows that the CC-check cannot vote to commit two transactions with a pairwise ARS conflict. □

Next, we show that the CC-check captures conflicts that render the ARS incomplete, *i.e.*, a row-key x that is in the PRS of T_j but should have been active. We assume that T_j's predicate set contains a predicate P that perceived x as passive. By definition of a conflict, however, T_i's write x_i fulfills P.

Lemma 15 *Pesto's CC-check detects incomplete ARS.*

We show this first for the unoptimized version of Pesto that does not leverage write monotonicity, before extending our proof to the general case. We note, that this unoptimized case is equivalent to a monotonicity grace period that is unbounded (or "infinite").

Lemma 16 *Pesto's monotonicity-unoptimized CC-check detects incomplete ARS.*

Proof. We again distinguish the two subcases: either the check for T_i was executed before the check for T_j or vice versa.

T_i **before** T_j**.** If T_i has passed the CC-check, and was committed, then T_i must either be in the *Prepared* or *Committed* set when the check is executed for T_j. Since monotonicity optimizations are disabled, it is guaranteed that the check for T_j explicitly compares with T_i when it reaches Line 12 in Algorithm 5. The check confirms (*i*) that T_i's write x_i fulfills P, (*ii*) that x is not present in T_j's ARS, and (*iii*) that there is no other transaction T_k with $ts_i < ts_k < ts_j$ that has prepared or committed a write x_k that does not fulfill P. This triggers the abort condition. Note that, if T_k exists but is only prepared, Pesto dynamically adds a dependency on T_k: if T_k aborts, T_j will abort too (Alg. 5, Lines 15-16 and Lines 21-24).

T_j **before** T_i**.** If T_j has passed the CC-check, and was committed, then T_j must either be in the *Prepared* or *Committed* set when the check is executed for T_i. When the check for T_i reaches Line 18 in Algorithm 5 it confirms that T_i's write x_i fulfills P, and x is not present in the T_j's ARS, triggering an abort.[5]

It follows, that the CC-check cannot vote to commit two transactions with a pairwise predicate conflict. □

[5]This check is more conservative than the T_i before T_j variant. If desired, it can be adjusted accordingly by comparing not only against concurrent reads, but dynamically checking whether there are other writers T_k that might render the conflict unnecessary

Next, we show that the CC-check remains safe when adjusted to use a finite monotonicity grace period. We omit a distinction of grace period tiers as they do not affect safety; they are equivalent to a single grace period and affect only efficiency.

Lemma 15 *Pesto's CC-check detects incomplete ARS.*

Proof. We need only expand the subcase in which T_i was executed before the check for T_j. The reverse case is unaffected by write monotonicity and already proven complete by Lemma 16.

T_i **before** T_j**.** Let ts_P be the table version of T_j's predicate P. We distinguish two subcases: (*i*) $ts_i \geq ts_P - grace$, and (*ii*) $ts_i < ts_P - grace$.

In case (*i*) no additional work need be shown, and we defer to Lemma 16. T_i will actively be considered for conflict when the check reaches Line 12 in Algorithm 5.

Case (*ii*) requires additional care: the abort condition in Line 12 of Algorithm 5 will not be triggered, yet we must ensure that T_i and T_j do not both commit. We show via contradiction that it is impossible for T_i to commit in case (*ii*).

Assume that T_i commits successfully. It follows from Pesto's commit protocol that at least $3f + 1$ (out of $5f + 1$) replicas voted to commit T_i and consequently prepared T_i. We know, further, that ts_P is the minimum table version observed across $3f + 1$ replicas, and, that T_j observed x_i at none of said $3f + 1$ replicas (since x_i is not in T_j's ARS). It follows from quorum intersection that at least one correct replica R_c prepares both T_i and computes T_j's ARS.

We distinguish two more subcases: (*i*) x_i was applied by R_c already, yet T_j did not read x_i, or (*ii*) x_i was not yet applied by R_c at the time of T_j's read.

Case 1 (*$write_i$* before *$read_j$*): Since $ts_i < ts_j$, and no other write x_k with $ts_i < ts_k < ts_j$ exists, x_i must be the latest version of x visible when $read_j$ executes. If $read_j$ omits inclusion of x_i into it's ARS, then either x_i does not fulfill P— a contradiction—, or x_i was only prepared and $read_j$ read from a snapshot and skipped past x_i. In the latter case, however, R_c would have dynamically adjusted its reported table version to be $ts_{P_c} \leq ts_i + grace$. Since $ts_P \leq ts_{P_c}$ it must be that $ts_i \geq ts_P - grace$ (case (*i*)), a contradiction.

Case 2 (*$read_j$* before *$write_i$*): Upon executing $read_j$, R_c adjusts its local montonicity threshold $ts_{mono} \geq ts_{P_c}$. There are once again two subcases. (*i*) $ts_{P_c} \leq ts_{mono} \leq ts_i + grace$. Since $ts_P \leq ts_{P_c}$ it must be that $ts_i \geq ts_P - grace$ (case (*i*)), a contradiction. (*ii*) $ts_{mono} \geq ts_{P_c} > ts_i + grace$. It follows from Line 3 of Algorithm 5 that the CC-check of T_i triggers the abort condition for violating write monotonicity. This is contradicts R_c voting to commit T_i (and preparing it locally).

It follows that the CC-check cannot vote to commit for two transactions with a pairwise predicate conflict. □

We conclude from Lemmas 14 and 15 that Pesto's CC-check returns a set of pairwise non-conflicting transactions, and thus Pesto fulfills Lemma 9.

Note: One can strengthen Line 18 in Algorithm 5 to abort a transaction only if $ts_P - grace < ts_i < ts_j$. The proof of safety follows analogously from the above proof. The monotonicity threshold and quorum interplay ensures that the above condition must hold for conflicting transactions if T_i commits.

Pesto adopts the commit (and fallback) protocol logic from Basil. Given Lemma 9, the proofs of Lemmas 10 and 11 consequently follow directly from Basil.

Finally, we show that an execution of Pesto satisfies *read validity*, *i.e.*, all reads from correct clients' committed transactions correspond to a committed write.

Lemma 17 *For any given execution of Pesto: all values read by correct clients' committed transactions were committed.*

Proof. By design, Pesto only makes prepared and committed writes visible. We thus must address only the case of reading prepared row-versions. It follows from Lemmas 12 and 13 that correct clients register a dependency for any prepared (row-key, row-version) pair in their active read set (ARS). Additionally, during the CC-check, a replica dynamically checks whether a passive row-version is prepared and whether an abort could reveal an active version that would render the ARS incomplete. If so, it adds a dependency for the prepared (passive) row-version.

It follows from Lines 21-24 of Algorithm 5 that a transaction only commits if all dependencies commit.[6] Consequently, all correct clients' committed transactions observe only committed writes. □

We conclude that Pesto upholds Byz-serializability:

Proof. Consider the set of transactions for which a C-CERT could have been formed. Consider a transaction T in this set. By Lemma 10, there cannot exist an A-CERT for this transaction. By Lemma 11, there cannot exist a conflicting transaction T' that generated a C-CERT. Consequently, there cannot exist a committed transaction T' in the execution history. The history thus generates an acyclic serialization graph. Finally, by Lemma 17, if T was issued by a correct client, then all reads of T were committed, and thus are explainable by the serial execution. The system is thus Byz-serializable. □

[6]Algorithm 5, line 5 further ensures that (Byzantine) clients cannot claim fabricated dependencies that may (intentionally) stall a transaction. This is not necessary for Byz-serializability, but ensures progress.

4.6.4 Byzantine Independence

Next, we show that Pesto upholds Byzantine independence. Since Pesto adopts Basil's point read and commit protocol, it suffices to show that Pesto's range read protocol does violate Byzantine independence.

Lemma 18 *Pesto's range read protocol upholds Byzantine independence.*

Proof. We distinguish the eager and snapshot execution paths.

Eager execution Byzantine independence follows directly from Lemma 12. All results are supported by a correct replica. Byzantine participants cannot take influence on the commit chance of a transaction. The result is, by definition, no more stale than a read to a single correct replica, and read sets (Q-READ), predicate sets (Q-PRED), and dependencies (Q-DEP) are backed by at least one correct client. A predicate's table version corresponds to the minimum reported version: a Byzantine replica can report an artificially small table version, but this affects only the efficiency of the CC-check, and not the outcome.

Snapshot execution The snapshot protocol introduces an additional layer of indirection. The snapshot proposal generation process requires that at least one correct replica vouch for every transaction in order to avoid proposing fabricated transactions. The proposal process is thus equivalent to a procedure that, for each row-key, consults with a single correct replica. Furthermore, proposing at transaction granularity ensures that every snapshot applied respects transaction atomicity; and thus the reading transaction is not subject to aborting due to reading from a non-serializable state.

Snapshot execution may require dynamic adjustment of table versions. Since adjustment at most makes the table version *smaller* this once again affects only efficiency and not the outcome of the CC-check.

Finally, snapshot-based execution—like eager execution—requires $3f + 1$ matching results, including consistent read and predicate sets as well as valid dependency sets. As a result, Byzantine indepdendence is preserved even for empty snapshots. □

Lemma 19 *Pesto upholds Byzantine independence in absence of a network adversary.*

Proof. Pesto adopts Basil's point read and commit protocol, and thus inherits its Byzantine independence in absence of a network adversary. It follows from Lemma 18 that Pesto's range read protocol upholds this property. □

4.6.5 Discussing Range Read Progress

Range reads in Pesto do not guarantee deterministic success. Range reads may fail (and need to retry) due to concurrent application of fresher commits at some replicas, or due to patchy snapshots (discussed next), causing replicas to read inconsistent versions.

Handling patchy snapshots

SS-VOTEs by default include only *id*s for the *freshest* row versions. This restriction, in combination with Pesto's snapshot filtering procedure, can have unintended consequences: if correct replicas are inconsistent, then filtering may eliminate all (transaction) candidates for a row, making it appear as if the row does not exist. In Figure 4.4, for instance, all replicas have observed a different subset of transactions (for a given key x): if replicas vote only with their latest version, then the resulting snapshot proposal is empty. To account for this, Pesto's execution procedure allows replicas to use their latest committed row as stand-in for any "missing" row.

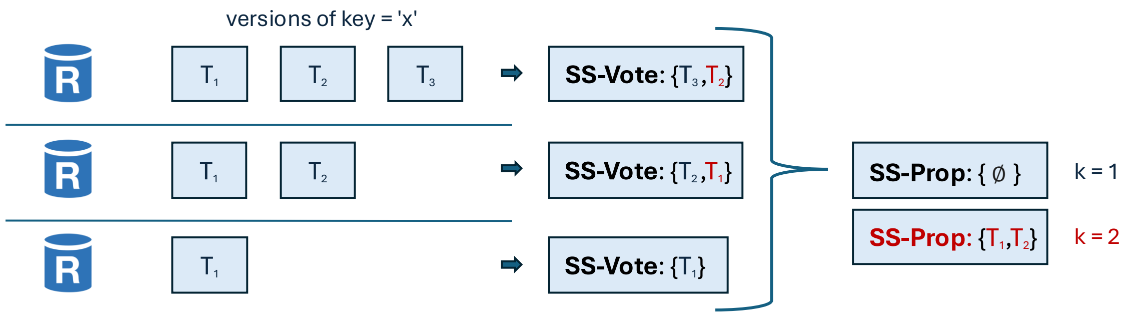

Figure 4.4: Snapshots including up to k = 1 (black) and k = 2 (red) versions per key

Additionally, replicas may opt to include the $k \geq 1$ latest versions of a given row (with k depending on the frequency a given row is written to), allowing the client to establish *some* recent common version. Figure 4.4 illustrates an example snapshot process for $k = 2$.

We note, that although snapshots can be patchy for small k (or extreme contention), this is not due to Byzantine influence. A snapshot quorum consisting entirely of correct replicas may produce insufficiently matching votes to successfully filter a transaction. Pesto requires at least $2f + 1$ snapshot votes to form a proposal, thus guaranteeing that even if Byzantine replicas (up to f) opt to fabricate their votes, the resulting snapshot proposal is no worse than a proposal sourced entirely from a subset of $(f + 1)$ correct replicas.

All roads lead to... Contention

Pesto allows replicas to favor reading newer committed versions over versions included in snapshots (or as stand-in for patchy snapshots). This may result in inconsistent execution results as some replicas might execute on the proposed snapshot, while others may execute on newer, locally processed committed versions. Although Pesto could strengthen the snapshot execution requirement to *force* consistency—and thereby ensure success of range reads—, this would be short-sighted as it fails to account for the holistic progress of a transaction. A transaction which succeeds in its read but accesses a stale version in the process may eventually have to abort, resulting in greater overall wasted effort. This approach effectively sacrifices the liveness of the overarching transaction to ensure the success of individual range reads.

Patchy snapshots are, likewise, an indicator for high contention. Candidate row-keys, for instance, might be dropped from a snapshot proposal because correct replicas differ in their latest observed versions and report only a small number (*e.g.*, $k = 1$) of versions per key. Raising k can help agree on a common version if there is inconsistency on the *prepared* versions, but, as discussed above, it is ultimately futile if replicas disagree on the latest committed versions.

To ensure the best possible *end-to-end* progress Pesto should strive to offer optimal freshness (to minimize missed writes, *i.e.*, conflicts) and to read in as few steps as possible (to minimize the conflict window opportunity [28]).

Pesto's snapshot freshness guarantee is configurable via the number of collected SS-votes. Requesting a quorum of $2f + 1$ SS-votes ensures that, for each (active) row-key, Pesto proposes a version no older than what would be obtained by querying a single correct replica (assuming unbounded k).

Clients can further enhance freshness by collecting $4f + 1$ SS-votes: This guarantees that the SS-prop includes the latest committed version known to *any* replica. This follows from the fact that every committed transaction must be prepared on at least $2f + 1$ correct replicas, of which at least $f + 1$ are guaranteed to be part of any quorum of size $4f + 1$. Since Pesto, when enhanced with write monotonicity, requires at least $3f + 1$ matching replies—and thus typically waits for up to $4f + 1$ replies during eager execution—this configuration can be adopted with little to no added cost.

By default, our prototype constructs an SS-PROP from $3f+1$ SS-VOTES, increasing the coverage of correct replicas while still allowing early progress when it appears unlikely that $3f + 1$ matching results will arrive. For improved freshness, this threshold can be raised to $4f + 1$, incurring only a small increase in client-side processing latency: the client already receives $4f + 1$ replies and must merely wait for and process them.

Finally, Pesto consciously favors freshness over consistency by opting to read fresher committed versions, in case the snapshot is stale. This ensures that, for any successful execution, the (active) read set is no more stale than a read to the committed state of a single correct replica.

Importantly, Pesto's range reads are always guaranteed to be responsive. Since snapshots cannot include fabricated transactions, synchronization, and consequently execution is guaranteed to be live. This ensures that a client reliably receives results. Failure to obtain matching results provides the client a signal about the level of contention. A client may choose to retry execution (possibly with larger k, and/or with larger snapshot quorums or read quorums), or (after a configurable amount of failures) choose to abort its ongoing transaction and retry with a new timestamp (possibly after backing off briefly). This contrasts with SMR-based designs, which always maintain the illusion of consistency and freshness—even when a transaction is ultimately doomed to abort.

Termination in absence of Contention

We briefly show that, in the absence of contention, Pesto's range read protocol ensures reliable termination. Informally, we say that a transaction contends with a read if it concurrently writes a value relevant to the read's query predicate. For simplicity, we assume the existence of a point in time, t_{CF}, after which the execution is contention-free. In practice, intermittent periods of low contention are sufficient in order to complete a range read.

Theorem 9 Pesto's range read protocol guarantees successful termination after t_{CF}.

Proof. Suppose all correct replicas exhibit the highest possible degree of inconsistency; *i.e.*, they differ on their latest (prepared) version for every row. By design, however, any committed transaction must have been prepared on at least $2f + 1$ correct replicas.

We assume that k is unbounded and that the client obtains $4f + 1$ SS-vote's. If this is not the case, a client may opt to retry with a more conservative configuration.

Since there are no concurrent contending writes, the resulting snapshot proposal is guaranteed to include the freshest committed row-version for every (active) row-key. All transactions in the snapshot proposal are available on at least one correct replica, allowing all replicas succeed in synchronizing all transactions in the proposal. Because no new contending transactions arrive, the snapshot captures a fully committed frontier and is complete (*i.e.*, not patchy), ensuring that all replicas read from the same state. As a result, all correct replicas produce consistent results, read sets, and predicate sets, ensuring successful termination. □

We note that, in most cases, replicas are sufficiently consistent to achieve success using a small k and smaller snapshot quorums. Replica consistency can be accelerated by employing lightweight gossip schemes that forward prepared and committed transactions.

Finally, we note that, in the absence of Byzantine clients (and contention), synchronization is not necessary as all correct replicas will eventually converge. In this case, simply retrying will eventually yield termination; snapshot synchronization merely accelerates the process. In the presence of Byzantine clients that intentionally (or just by crashing) disseminate their transactions to only a subset of replicas, however, eventual consistency is not guaranteed. The snapshot protocol consequently serves also as a means to ensure reliable termination of incomplete transactions.

Termination does not imply Commit

Successful range read execution does not imply that the overarching transaction will commit. A conflicting write may arrive after the read completes, but before the associated transaction tries to commit—resulting in an abort. This is inevitable in presence of contention. We thus once again limit our discussion to the period after t_{CF}—the point after which no more contending transactions arrive.

Pesto's range read protocol (with sufficiently large quorums and k) ensures that a range read will read the freshest *committed* version for any given row. It is possible, however, that execution will miss fresher *prepared* versions. Consider, for instance, a prepared version (perhaps issued by a Byzantine client) that is only replicated to at most $2f$ correct replicas. Even a snapshot quorum of size $4f + 1$ might not include the version's transaction id $f + 1$ times, resulting in exclusion from the proposal. This may cause the read of a stale (active) version, ultimately resulting in the reading transaction's abort.[7] Successfully committing transactions thus requires synchronizing also the (freshest) prepared versions. Pesto relies on two practical mechanisms to do so.

First, transactions that abort due to conflicts with prepared transactions trigger the fallback protocol. The client of aborting transaction T will try to commit the conflicting

[7]Note that if the prepared transaction is only prepared at a few replicas then it might be possible for the reading transaction to succeed in assembling a Commit Quorum; in this case it is the prepared transaction that will ultimately abort, and not the reader! No additional coordination is necessary.

transaction T' itself. This ensures that (*i*) all replicas receive the transaction (and thus become consistent), and (*ii*) T' actually terminates, and thus any acquired dependency is reliably resolved.

Second, Pesto replicas may gossip prepared transactions. This can be done either eagerly, upon first receiving a transaction; or lazily, upon observing inconsistency for range reads (a replica may then opt to gossip the transactions for the row-versions it considers active).

4.7 Evaluation

Our evaluation seeks to answer the following questions:

- How does Pesto perform on realistic applications? (§ 4.7.1)
- How does Pesto compare to Basil's KVS design? (§ 4.7.2)
- What is the impact of inconsistency on Pesto? (§ 4.7.3)
- How well does Pesto tolerate replica failures? (§ 4.7.4)

Implementation We implement a prototype of Pesto in C/C++, starting from the open source implementation of Basil [170]. We use Protobuf [75] and TCP for networking, ed25519 elliptic-curve digital signatures [19, 134] and HMAC-SHA256 [43] for authentication, and Blake3 [177] for hashing. For its query layer, Pesto adapts Peloton [77], a full fledged open-source SQL Database based on PostgreSQL [78]. We modify Peloton to use Pesto's Concurrency Control and support snapshot synchronization; we remove Peloton's self-driving configuration features, and make several optimizations to its index selection and execution procedures.

Baselines We compare against four baseline designs:

1. Unreplicated Peloton, run in-memory on a dedicated server. We adopt our non-Pesto specific optimizations for a fair comparison.
2. Peloton-SMR, a strawman system that layers Peloton atop BFT state machine replication (SMR). We layer Peloton atop HotStuff (Peloton-HS) [192]—a popular BFT consensus protocol that forms the basis of several commercial systems [10, 13, 31, 60, 179, 181]—, and BFT-SMaRt (Peloton-Smart) [23, 176], a state-of-the-art PBFT-based [30] implementation.

   For correctness, SMR-based designs require deterministic execution on each replica: this requires either sequential execution (which drastically limits performance) or implementation of complex and custom parallel execution engines [49, 57, 68].For maximum generosity to the baselines, we opt to relax the determinism requirement for Peloton-SMR: we allow replicas to freely execute transactions in parallel, and designate a "primary" replica to respond to clients to ensure serializability. This system configuration is explicitly not fault tolerant, but simulates the optimal upper-bound on performance. Finally, we augment both Peloton-SMR prototypes to benefit from Basil's reply batching scheme [171] to amortize signature generation overheads.
3. Third, we compare against PostgreSQL [78] (commonly referred to as *Postgres*), a production-grade SQL database. We run Postgres both unreplicated, and with its native primary backup feature (*Postgres-PB*) where writes are synchronously replicated (and written to a WAL), mounted in memory using tempfs.
4. Finally, since Peloton and Postgres are not easily shardable, we also compare Pesto against CockroachDB (CRDB) [37, 175], a popular *distributed* database of production grade. Because CRDB has poor single node performance (it's CPU

utilization and query processing latency are much higher than Peloton/Postgres) we instantiate it with 6 shards (one machine per shard). We run CRDB unreplicated, in-memory.

Table 4.1 summarizes all evaluated systems.[8]

Baseline	Description
PESTO	Our system. A BFT database that is SQL compatible and shardable.
PESTO-UNREP	A toy variant of Pesto that is unreplicated. Only used for microbenchmarking purposes.
PELOTON	An unreplicated (non-fault tolerant) SQL database. Peloton [77] forms the basis for Pesto's database engine.
PELOTON-HS	A BFT database built by layering Peloton atop HotStuff [192], a BFT consensus protocol.
PELOTON-SMART	A BFT database built by layering Peloton atop BFT-SMaRt [176], a BFT consensus protocol.
POSTGRES	A popular unreplicated SQL database of production grade [78].
POSTGRES-PB	A deployment of Postgres using its native primary backup replication (one backup replica).
CRDB	A popular distributed SQL database of production grade [37]. We instantiate CRDB with 6 shards.

Table 4.1: Summary of evaluated systems.

Experimental Setup We use `m510` machines (8-core 2.0 GHz CPU, 64 GB RAM, 10 GB NIC, 0.15 ms ping latency) on CloudLab [36]. Clients execute transactions in a closed-loop, and reissue aborted transactions using a standard random-exponential back-off scheme. We measure transaction latency as the time difference between initial invocation and commit. We configure each replicated system to tolerate $f = 1$ faults ($n = 3f + 1$ for Peloton-SMR, $n = 5f + 1$ for Pesto, $n = 2$ for Postgres-PB); Peloton, Postgres and CRDB are run unreplicated and tolerate no faults. We run experiments for 60 seconds, including a 15 s warm-up and cool-down period.

[8]Our system prototypes are available at `https://github.com/fsuri/Pequin-Artifact`.

4.7.1 High level performance

We evaluate Pesto on three popular transactional benchmark applications: TPC-C [182], AuctionMark [50], and SEATS [50]. TPC-C simulates the business of an online e-commerce application. We configure it to use 20 warehouses, and instantiate indexes to retrieve orders by customer, as well as customers by their last name. TPC-C exhibits high contention (*e.g.*, all NEW-ORDER transactions on same warehouse conflict), and a high ratio of point to range reads. AuctionMark models an online auction platform, while SEATS simulates an airline reservation system. Both workloads are characterized by a high fraction of range queries and cross-table joins, but they exhibit overall low contention relative to TPC-C. We model our implementation after Benchbase [50, 76]; we instantiate both workloads with a scale factor of 1.

Figures 4.5, 4.6 and 4.7 present the results for TPC-C, AuctionMark, and SEATS, respectively.

TPC-C Pesto's throughput (1784 tx/s) matches that of unreplicated Peloton (1777 tx/s) and Postgres (1781 tx/s), is 2.3x higher than that of Peloton-HS (758 tx/s) and Peloton-Smart (785 tx/s), and 1.4x higher than Postgres-PB (1257 tx/s). Pesto increases latency by less than 1.5x over Peloton and Postgres (equal latency at high load), and reduces latency by 3.9x over Peloton-HS, and 2.7x over Peloton-Smart. Peloton-HS and Peloton-Smart incur the latency of consensus (3 message delays (md) BFT-SMaRt, 7 md HotStuff) for each read, write and commit request; Postgres-PB incurs replication latency for each write, but performs reads at the primary only. Pesto, in contrast, (*i*) buffers writes, (*ii*) executes point reads in a single round trip as well as 99.9% of range reads, and (*iii*) can commit in a single round trip (Fast Path) 97% of the time. While Pesto, Peloton and Postgres remain CPU bottlenecked, both Peloton-SMR systems and Postgres-PB are contention bottlenecked due to their higher latency, thus limiting their

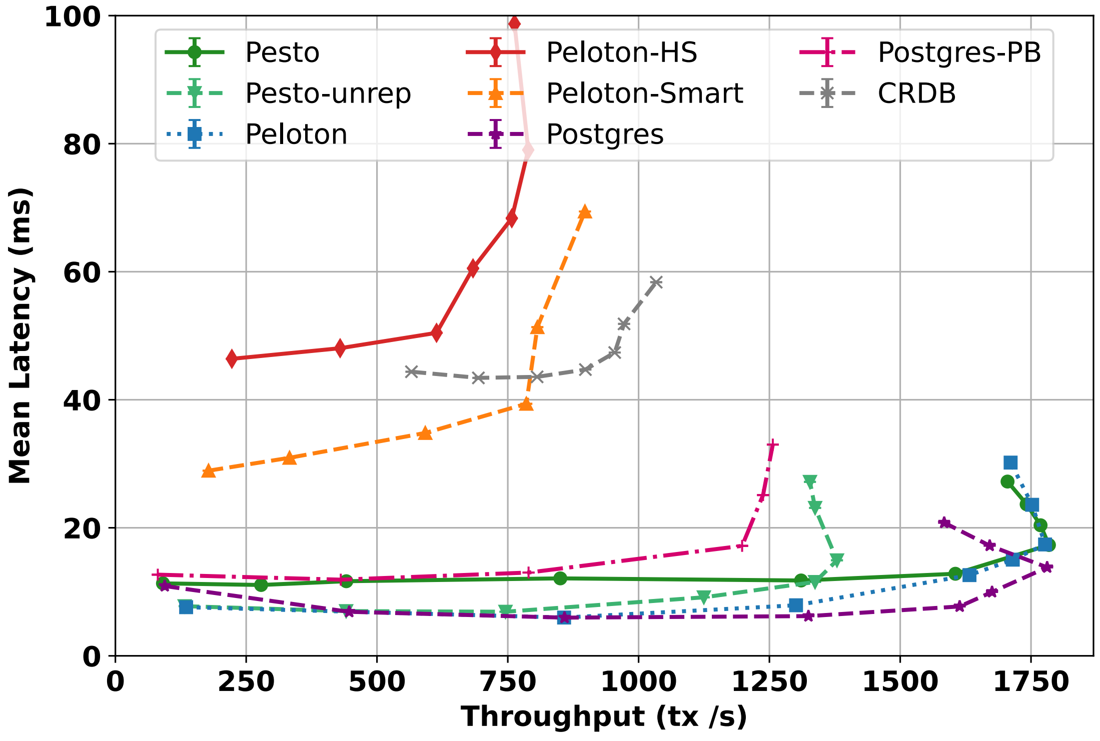

Figure 4.5: TPC-C (20wh) Throughput vs. Latency

achievable throughput. CRDB, too, is contention bottlenecked, peaking at 1033 tx/s; although CRDB does not replicate, it supports only sequential reads within each transaction which results in high latency on TPC-C.

Perhaps surprisingly, Pesto matches the throughput of unreplicated Peloton *despite* the overheads inherent to BFT protocols (*e.g.*, signatures and quorum requirements). This is because Pesto must read only from a quorum of replicas (at least $f + 1$ for point reads, and at least $3f + 1$ for range reads): on a point read heavy workload as TPC-C this allows Pesto to efficiently load-balance read requests and exceed its unreplicated performance. Unreplicated Pesto is able to closely match Peloton in latency, but reaches a CPU bottleneck at 1379 tx/s.

AuctionMark and SEATS make fewer point reads which diminishes the benefits of request load balancing (range reads require larger quorums). Nonetheless, Pesto is able

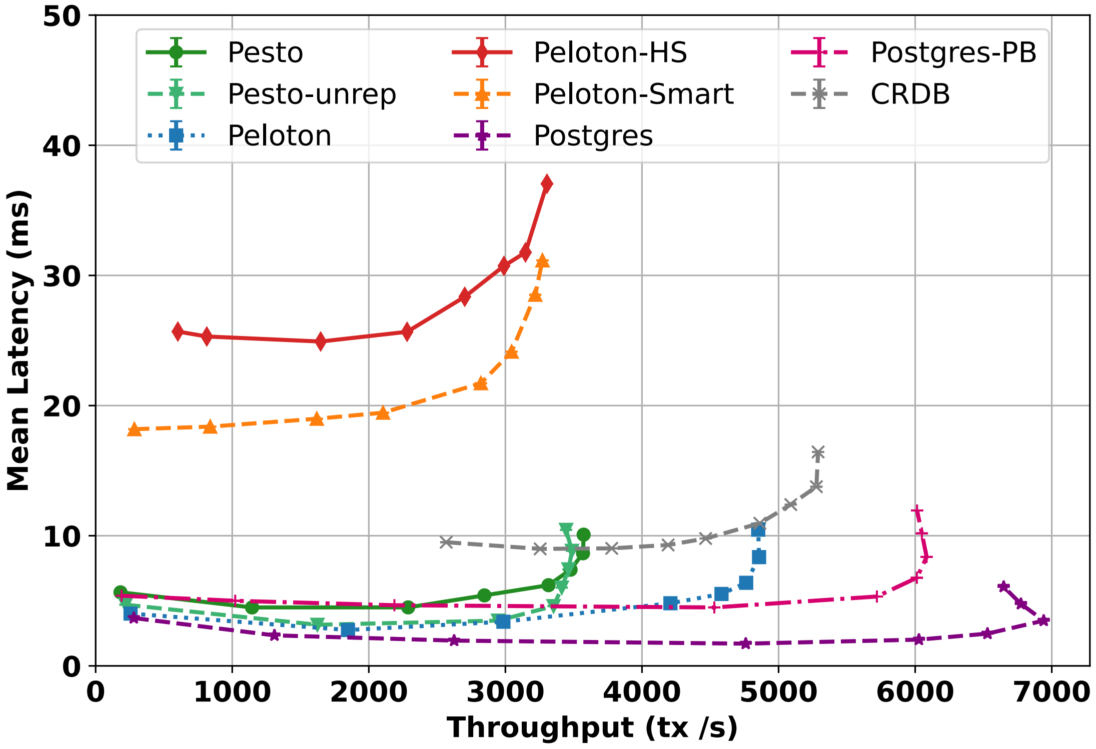

Figure 4.6: AuctionMark Throughput vs. Latency

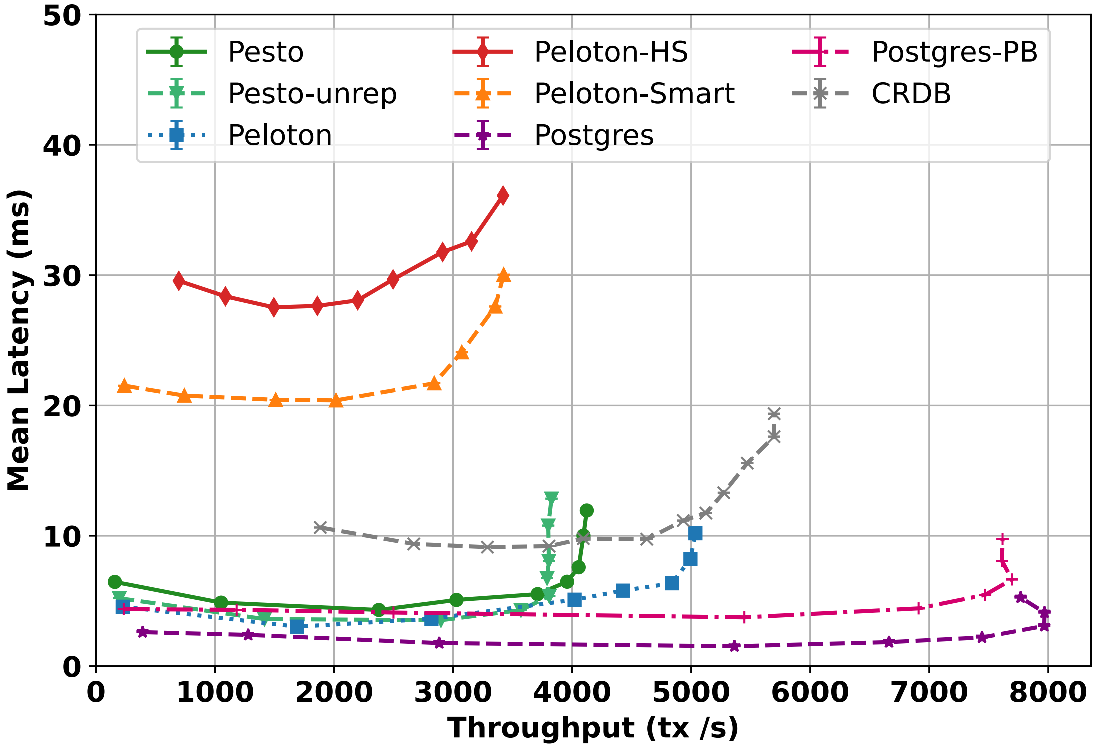

Figure 4.7: SEATS Throughput vs. Latency

to match its unreplicated throughput, while coming within 1.36x of unreplicated Peloton on AuctionMark, and 1.22x on SEATS; Pesto comes within 1.94x of Postgres on both workloads. Pesto reduces latency over Peloton-HS and Peloton-Smart respectively by 5x/3x on AuctionMark, and 4.6x/3.4x on SEATS. Throughput gains are limited (1.1x AuctionMark, 1.2x SEATS) as all systems are CPU bottlenecked.

Takeaway Pesto achieves performance comparable with unreplicated production-grade systems, while significantly outperforming traditional BFT-based approaches.

4.7.2 Comparison with Basil's key-value store design

Next, we examine the overheads and benefits introduced by Pesto, compared to Basil's key-value store-based approach.

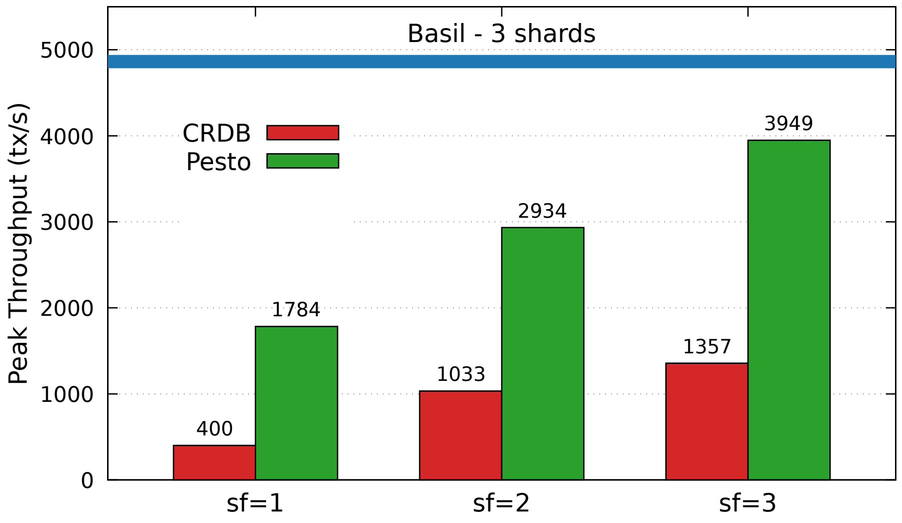

Figure 4.8: Scalability with increasing number of shard on TPC-C (20wh)

Scalability Figure 4.8 shows the scalability of Pesto on TPC-C with increasing number of shards. Pesto is CPU bottlenecked and thus scales significantly by partitioning the workload across two (1.64x) and three shards (2.21x), respectively. On a shared three-shard setup, Pesto—which implements a full-stack SQL system, requiring significant CPU cycles for query parsing, planning, execution, and index management—comes within 1.23x of the reported throughput of Basil (4862 tx/s), which implements only a simple key-value store. We also compare Pesto's scalability to CRDB. Because Pesto uses more machines (for replication), we allow CRDB to scale to 6 and 9 shards (its peak). CRDB's has poor single-shard performance (4.46x less throughput than Pesto), but scales at a rate similar to Pesto. Its peak performance is 2.91x below Pesto.

Range vs. Point Reads While Pesto generally incurs overhead relative to Basil, this is not the case for all transactions. Pesto's range read protocol reduces the latency of TPC-C's scan-heavy `Stock-level` transaction by over 11x compared to Basil's point-read based implementation.

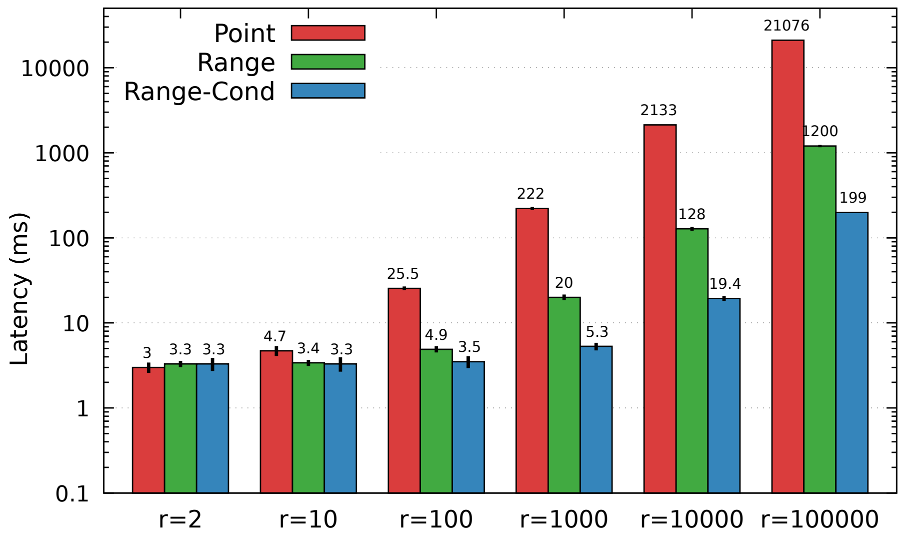

Figure 4.9: Latency comparison for the same query implemented using point reads versus range reads, as the number of accessed rows increases.

We illustrate the benefits of Pesto's range read protocol in Figure 4.9, which reports scan latency across varying ranges on a simple read-only microbenchmark. As more rows are accessed, the latency of a point-only implementation increases significantly due to the need to process and verify messages based on the size of the *intermediary* *result*. In contrast, range reads scale significantly better (a 16.6x reduction for a range of 10k rows), with only a single message exchange required for the entire query. The cost of range reads scales with the *result* size. For example, if a scan's result is conditioned on a predicate that holds for only 1 in 100 rows, range reads scale accordingly (a 110x reduction for a range of 10k rows).

4.7.3 Stress testing Range Reads

Range reads offer improved expressivity and performance but might not succeed in a single round trip. To evaluate the worst-case, we stress test Pesto by (*i*) artificially failing eager execution for *every* transaction (requiring a snapshot proposal, but no synchronization), and (*ii*) artificially simulating inconsistency by ommitting/delaying writes of *every* transaction at $\frac{1}{3}$rd of replicas (requiring also synchronization).

We implement a microbenchmark based on YCSB [39] consisting of 10 tables, each containing $1M$ keys. Every transaction reads and updates 10 rows. We distinguish two workload instantiations: an uncontended uniform access pattern U, and a very highly contended Zipfian access pattern Z with coefficient 1.1. Figure 4.10 shows the results.

On the uniform workload Pesto is CPU bottlenecked. Failed eager execution (*U-FailEager*) incurs an extra round trip to propose a snapshot and re-execute on the synchronized state. Since each transaction executes *twice*, CPU load increases, reducing throughput (≈9%) and increasing latency (1.38x). Inconsistency (*U-Incon*) yields similar results: two thirds of transactions fail eager execution and require both a snapshot proposal and synchronization between replicas to exchange missing writes. However, this cost is partially offset by the initial omission of writes at one third of replicas, resulting in an overall throughput reduction of only $\approx 5\%$.

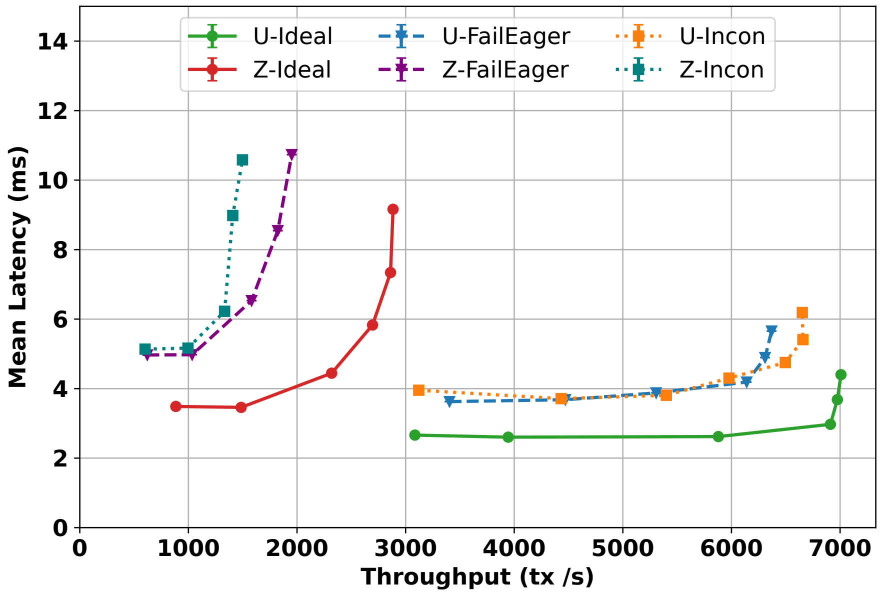

Figure 4.10: Stress testing range reads under simulated inconsistency

The Zipfian workload, in contrast, induces a heavy contention bottleneck. The respective up-ticks in read latency for *Z-FailEager* (1.43x) and *Z-Incon* (1.49x) increase the opportunity for conflict (*contention windows* grow [28]), resulting in a rise of transaction aborts. Throughput drops by 32% and 48% respectively.

4.7.4 Impact of Failure

Finally, we evaluate the impact of replica failures in Pesto. Note that replicas cannot impact the correctness or liveness of Pesto's range read protocol. The snapshot filtering procedure ensures that all proposed transactions are valid (and thus can be reliably synchronized), and no more stale than a read to any single correct replica. Similarly, replicas cannot affect the safety of Pesto's commit protocol (this follows from Basil). Replicas can only impact the system by crashing.

Figure 4.11 shows the effect of $f = 1$ failures on the Uniform and Zipfian microbenchmarks. Crucially, and unlike SMR-based designs that rely on a leader [30, 73, 101, 192], Pesto suffers *no progress interruptions* as transaction coordination is entirely client driven. Replica failures affect only Pesto's ability to commit in a single round trip (*fast path*).

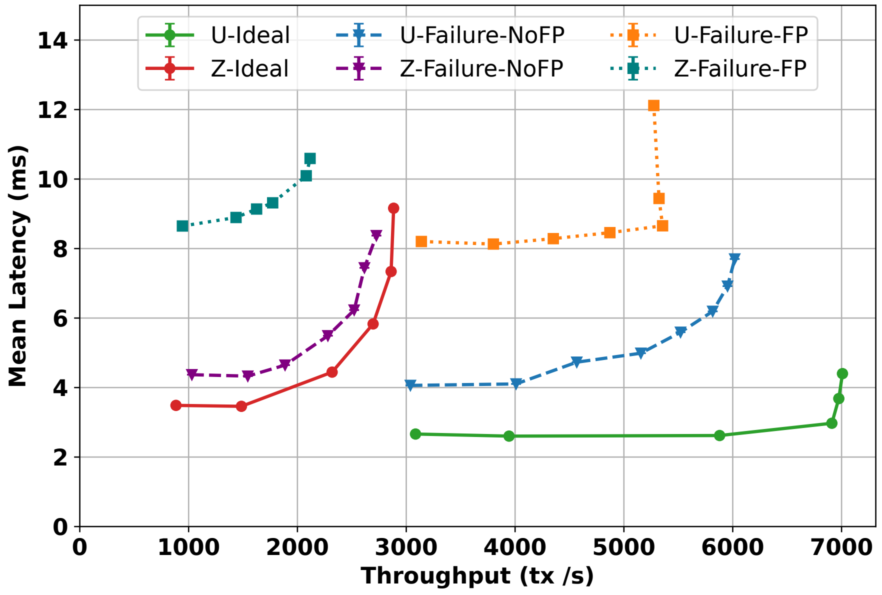

Figure 4.11: Performance impact of a replica failure

We evaluate two configurations: (*i*) *Failure-NoFP* illustrates the effect of a failure when the fast path is disabled. (*ii*) *Failure-FP* shows the impact of a failed fast path when using a very conservative timeout of ≈ 4 ms. In principle, fast and slow path execution can run in parallel to avoid timeout-induced delays. However, this introduces redundant processing when transactions succeed on the fast path. By default, Pesto delays the slow path until a timeout to optimize resource efficiency.

In both configurations, commits require an additional round trip of coordination (to a single shard, § 4.5.2) to ensure durability. This increases latency, and, for the CPU bottlenecked uniform workload, reduces throughput due to added signature overhead. *U-Failure-NoFP* and *U-Failure-FP* degrade throughput by 14% and 24%, respectively, while latency increases by 1.59x and 2.7x.

In contrast, on the contention bottlenecked Zipfian workload the slow path overhead only marginally impacts throughput and latency (1.25x latency increase, and a 5% throughput reduction for *Z-FailureNoFP*). This is a direct consequence of Pesto making writes visible eagerly upon preparing and allowing contending transactions to acquire dependencies instead of waiting for commitment. The additional slow path latency is incurred only *after* preparing, and thus leaves conflict windows mostly unaffected. *Z-FailureFP* incurs the additional timeout latency (2.5x), and reduces throughput by 36%.

We defer a detailed analysis of client failures to our evaluation of Basil (§ 3.6). Client failures affect *only* commit liveness—not the commit outcome—and are resolved via Basil's cooperative fallback protocol, which Pesto adopts. Client failures *before* commit affect only itself, as its writes are not yet visible; clients can only impact the execution of their *own* queries, and thus cannot affect the correctness or progress of correct clients' executions.

4.8 Extended Technical Discussion

This section outlines supplementary technical details, optimizations, and considerations for Pesto beyond those discussed in the main technical sections.

4.8.1 Impact of Byzantine Timestamps

Each transaction in Pesto is assigned a unique timestamp ts_T that implicitly establishes the final serialization order of transactions. This timestamp is generated client-side as

$ts := (localtime, ClientID, seq\text{-}no)$. Byzantine clients may freely select arbitrary timestamps; here, we discuss briefly the implications of such behavior.

Choosing a timestamp that is artificially small has minimal impact. Reads will simply access an older snapshot version—or be rejected if such versions have been garbage collected (see § 4.8.2). Writes with low timestamps tend to abort during validation because they would invalidate existing prepared or committed reads with larger timestamps.

Conversely, choosing artificially large timestamps is more problematic. While writes with future timestamps cause no immediate issues—they are simply stored "in the future"—reads can create extended conflict windows. Specifically, any concurrent writes with timestamps between the version read by the reader and the reader's own timestamp must abort, as they would otherwise invalidate the reader's snapshot.

To mitigate abuse by Byzantine clients fabricating excessively large timestamps, replicas reject transactions whose timestamps exceed their local clock R_{Time} by a threshold δ, which accounts for client ping latency and clock skew. For a Byzantine client to induce conflicts, it must successfully prepare or commit its transaction; aborted or rejected transactions remain invisible and thus do not affect correct concurrent clients. Therefore, to maximize the probability of committing, it is rational—even for Byzantine clients—to choose timestamps that accurately reflect real time. While Pesto does not rely on δ for safety or liveness, a well-chosen value can improve the system's throughput.

4.8.2 Multi-version Garbage Collection

Writes in Pesto create new row versions indexed by the writing transaction's timestamp. To enable garbage collection of old versions, Pesto—like Basil—enforces a timestamp bound on readable and writeable versions. Each replica maintains a low-watermark *gc*, which lags behind its local clock and marks the cutoff point. New reads or writes with timestamps below *gc* are ignored. When writing a new row version with $TS > gc$, the replica dynamically deletes all but the latest version with timestamp smaller than *gc*. This ensures that valid readers with read timestamps $TS \geq gc$ can find a readable version. To clean up rarely-updated rows, replicas may also perform periodic sweep rows to garbage collect old versions.

This scheme bounds the age of stored writes, but not their frequency. A highly contended key, for instance, may receive many writes in quick succession; all with timestamps above *gc*, preventing their immediate garbage collection. To prevent storage bloat (*e.g.*, to avoid abuse by an authenticated Byzantine client) Pesto can employ two additional mechanisms. (*i*) First, Pesto may rate-limit clients to only one active transaction at a time. This limits write frequency, and additionally ensures that Byzantine clients must complete prior transactions before issuing new ones, minimizing transaction stalls in the system. (*ii*) Pesto may enforce a per-key version limit *l* (this may be configured differently for different objects). A correct replica can delete all but the freshest *l* versions (with timestamps greater than *gc*), and reject any reads to the key with timestamps older than the *l*th version. If a concurrent read transaction depends on a (prepared) version has beem garbage collected (*i.e.*, older than *l*th version), that transaction is aborted. This is safe and sensible, as the transaction would likely abort anyway due to reading stale data.

If a snapshot-based read—*i.e.*, a range read applying an SS-PROP—requests a version that has already been garbage collected, the replica returns the freshest available version instead. While this may reduce the number of matching read replies, it does not compromise safety and often aids transaction progress, since garbage-collected versions are (*i*) typcally stale and (*ii*) would otherwise cause dependent transactions to abort.

4.8.3 Active Snapshots

Pesto optimistically records in its read sets and snapshot votes only the metadata of those rows that are relevant to the query computation. Specifically, an active snapshot includes the transaction *id*'s associated with row versions that satisfy a query's filter predicates—*i.e.*, the *active* rows.

Record relevant *passive* versions Including only the *id*s of active rows—that is, rows whose freshest version satisfies the query predicate—greatly reduces snapshot size but can cause snapshots to be unnecessarily stale. For example, if some replicas do not include a row-key because their latest version v does not fulfill the predicate (*e.g.*, $k = 1$), while others include an earlier version $v' < v$ that does, Pesto may reconstruct a stale state (v'). This can cause the reading transaction to abort at commit time due to missing a newer version. To prevent this, replicas should also include in snapshots those fresher versions for which a past version satisfies the predicate. We call these included (passive) versions *active-negative*.

Nested Queries Active snapshots may require additional care when handling complex or nested queries. Consider the nested query `SELECT * FROM` t_x `WHERE x > (SELECT MAX(y) FROM` t_y`)`: the active rows of the outer query Q_o depend on the result of the inner query Q_i. Since replicas' SS-VOTES may differ, the resulting (potentially patchy) SS-PROP may not be commutative with a sequential snapshot execution of Q_i followed by Q_o. In such cases, it may be advisable to either (*i*) use coarser snapshots, or (*ii*) rewrite nested queries into sequential patterns. In practice, however, replicas tend to remain highly consistent and execution typically succeeds even on the eager path (§ 4.7).

A note on freshness Active snapshots may, in rare edge cases, lead to slight freshness degradation. If some (but not all) correct replicas observe *only* passive row-versions among their latest k versions and therefore omit the row-key from their SS-vote, the resulting snapshot may reflect a version that is more stale than what a single replica could return. Notably, if the row-key is omitted entirely, this is equivalent to materializing a passive row-version, which is the desired outcome.

Consider the following example with a snapshot quorum of $2f + 1 = 3$ SS-votes (assuming $f = 1$) for a key x with two versions: x_1, which satisfies the query predicate P, and x_2, which does not. Replica R_1 has committed x_2 and sees no active version, so it casts SS-vote$_1$:= {}. Replica R_2 has committed x_1 and prepared x_2. It casts SS-vote := $\{x_1, x_2\}$, including x_2 as an *active-negative* version. Replica R_3 is Byzantine and votes at whim: it casts SS-vote := $\{x_1\}$. The resulting snapshot proposal is SS-prop := $\{x_1\}$, even though both correct replicas have observed the fresher x_2. Because R_1 omitted x_2, the snapshot reflects an unnecessarily stale state. In effect, freshness degrades from the equivalent of reading the freshest version from a single correct replica, to selecting the k-freshest version from a single replica. If instead SS-vote$_3$:= {}, then the resulting SS-prop is empty—effectively the same as reading x_2, which is passive and does not affect P. This is a valid and even desirable outcome.

Importantly, this edge case does not compromise Byzantine independence. While the Byzantine replica may influence which version is selected, it cannot dictate the outcome. The scenario is no worse than if the faulty replica had abstained from voting entirely, and a different correct replica had reported only x_1 as its latest active version.

4.8.4 Snapshots with Optimistic Transaction IDs

Snapshots can be further compressed by optimistically replacing transaction identifiers with transaction timestamps. Unlike transaction identifiers—which are statistically independent cryptographic hashes (256b)—timestamps are smaller (64b) and temporally correlated, allowing more efficient encoding: a simple delta encoding can reduce them to 32b, and integer compression can shrink them further to under 16b.

However, Byzantine clients may equivocate by assigning the same timestamps to two distinct transactions. This can cause snapshots to diverge: upon receiving a snapshot proposal containing timestamp ts_T, two correct replicas may associate it with different transactions T and T' ($ts_T = ts_{T'}$), leading to inconsistent synchronization. Fortunately, this is a low-yield and easily detectable attack. Since timestamps embed client identifiers and all transactions are authenticated, any client that reuses a timestamp across different transactions is explicitly identifiable. Replicas that detect such behavior can report and exclude the faulty client from further participation. To improve robustness, a correct client whose snapshot execution fails may retry using standard transaction identifiers, which are globally unique and therefore immune to ambiguity.

4.8.5 Distributed Queries

Orchestrating and implementing cross-shard query *execution*—such as scans or joins over partitioned tables—is well-studied [14, 99, 175, 193] but non-trivial, and beyond the scope of our prototype. These challenges are not unique to Pesto, and arise similarly in SMR-based systems.

We briefly outline how to coordinate Pesto's snapshot protocol across shards. For simplicity, we assume each shard includes a replica operated by the same trust authority (*i.e.*, every replica has a trusted counterpart in other shards). We leave the exploration of efficient, trust-free cross-shard execution to future work.

We distinguish between *flat* and *nested* queries. Simple flat queries, such as range scans (*e.g.*, `SELECT * FROM tbl WHERE x > 5`) or hash joins [133], require no cross-shard coordination during snapshotting. Each shard can independently compute an SS-PROP by executing the query locally. Once all involved shards have synchronized, cross-shard query execution proceeds. One shard may act as a coordinator to aggregate the final results. If the client is unaware of the partitioning scheme, it suffices to contact the coordinator, which forwards the request to the appropriate shards. Read sets, predicate sets, and dependency metadata can be collected directly from each individual shard.

Nested queries, by contrast, may require cross-shard execution *during* snapshotting due to dependencies between inner and outer sub-queries. As noted in Section 4.8.3, such dependencies can complicate snapshotting even within a single shard. Sharding can amplify this challenge, as intermediate sub-query results may determine which shards to involve—potentially leading replicas operated by different trust authorities to involve inconsistent shard sets. In such cases, it may be advisable to rewrite queries into sequential stages, or conservatively expand the snapshot scope to include all potentially relevant shards.

These complexities are not specific to Pesto: in SMR-based systems as well, nested cross-shard queries may require sequential coordination to determine the involved shards, and each sub-query must be replicated consistently via consensus.

4.9 Related Work

In addition to Basil (Chapter 3), the BFT key-value store Pesto builds upon, there are several other related research efforts.

BFT State Machine Replication State machine replication (SMR) [160] provides the abstraction of a single fault tolerant server, a core building block in many distributed data-storage systems, both in the Crash Fault Tolerant (CFT) [14, 40] and BFT space [7, 8, 16, 98]. At the heart of BFT SMR lie consensus protocols [30, 35, 73, 80, 101, 165, 192] enable replicas to establish a consistent total order of requests, despite arbitrary misbehavior. This powerful abstraction, unfortunately, does not come cheap.

Reaching agreement requires several rounds of message exchanges, resulting in high latency. To facilitate agreement BFT consensus protocols traditionally designate a *leader* replica to act as a designated sequencer [30, 80, 101, 192]; this marks a scalability bottleneck and raises fairness (and censorship) concerns [197]. Recent works propose multi-leader approaches [46, 73, 165, 166] that improve throughput and fairness at the cost of increased latency. Pesto, following in Basil's footsteps, sidesteps both performance and fairness concerns by adopting a client-driven (leaderless) approach and enforcing Byzantine independence.

To maintain consistency, replicas in SMR must further execute requests sequentially, limiting scalability; though some works explore ways to regain limited parallelism [49, 57, 68]. Pesto, in contrast, is order-free by design, and naturally parallelizes concurrent executions. Finally, SMR-based systems, by default, require that all replicas execute every operation. Yin et al. [190] and Distler et al. [51] explore separating agreement from execution to reduce redundancy; Pesto, likewise, need only execute at a subset of replicas, enabling load balancing.

Database functionality can be layered on top of SMR (or vice versa) [94, 159], but at high cost.

Blockchains [60, 62, 64, 65] offer neither interactive transactions nor SQL, and instead implement custom *smart contract* (SC) languages [198] (effectively stored procedures); SC invocations are ordered using BFT SMR and executed by native engines such as the Ethereum Virtual Machine [63] or Move runtime [61].

DB atop BFT BlockchainDB [55] layers a DB atop existing blockchains and shards contents across peers to reduce replication redundancy; however, it does not implement transactions and offers only a `GET/PUT` interface. BigchainDB [127] implements a custom NoSQL [83] interface and layers MongoDB [88] on top of Tendermint [27]. FalconDB [151] leverages authenticated data structures to allow clients to safely execute limited SQL queries against a single replica; it orders transaction commits via Tendermint and uses OCC to enforce snapshot isolation. The Blockchain Relational Database [139] and Kwil [180] layer PostgreSQL [78] atop BFTSmart [176] and CometBFT [178], respectively, but limit SQL transactions to stored procedures.

BFT atop DB Hyperledger Fabric [8] adopts an Execute-Order-Validate framework: stored procedures (Chaincodes) are executed optimistically in parallel across replicas (peers), and ordered for validation. ChainifyDB [161] implements a similar architecture but supports a general purpose SQL interface and allows replicas to deploy heterogeneous relational DB's. Transactions are executed optimistically, and attempt to reach agreement on *results*; if executions are inconsistent, database states are rolled back, and transactions re-executed.

SemanticCC Pesto's SemanticCC builds on the principles of predicate and precision locking [56, 92]. Hekaton [48], too, leverages semantics to avoid aborts, but does so by tracking the full read set and re-executing a transaction during validation to detect missed versions. HyPer [140] adapts precision locking to optimistic concurrency control (OCC), but only in the context of an unreplicated database. In contrast to HyPer, which stores predicates server-side during execution, Pesto stores no query metadata during execution and instead relies on clients and write monotonicity to enforce serializability.

4.10 Conclusion

This chapter introduced Pesto, a high performance BFT database that provides a general SQL purpose interface. Pesto forgoes explicit ordering of requests, allowing execution to proceed in parallel, and with low latency. It implements Byz-serializable transactions and upholds Byzantine independence, thereby limiting the influence of Byzantine participants.

CHAPTER 5
CONCLUSION

This dissertation presents a novel approach to building high performance Byzantine fault tolerant (BFT) data storage systems for decentralized applications. Drawing on the principles of classical, crash fault tolerant (CFT) distributed databases, we challenge the conventional reliance on implementing a shared, tamper-proof totally ordered log via State Machine Replication (SMR). Instead, we advocate for enforcing only *serializable* executions—that is, executions equivalent in effect to some total order—thus avoiding the costly overhead of total ordering. This shift enables both improved performance and enhanced robustness.

We realize this approach through the design of two systems: Basil [171] and Pesto [172]. Basil serves as the foundation, integrating replication and distributed transaction coordination into a single, low-latency storage layer. Its order free, client-driven architecture allows transactions to execute independently and in parallel, as long as they do not conflict—achieving higher throughput, lower latency, and improved fault isolation compared to traditional SMR-based designs. Building on Basil, Pesto adds support for rich, SQL-style query interfaces, preserving performance while enabling seamless integration with existing application stacks.

Together, Basil and Pesto demonstrate that it is possible to scale the abstraction of a shared, totally ordered log by avoiding explicit ordering and coordinating only when necessary—thus opening a new design space for more efficient and expressive BFT transaction processing.

5.1 Other Work

This section briefly highlights additional research projects to which I contributed to substantially during my PhD, but which are not included in this dissertation.

5.1.1 Autobahn: Seamless high speed BFT

Autobahn [73] is a general purpose Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) architecture that delivers low latency, high throughput, and improved resilience to intermittent progress interruptions.

The Quest for Seamlessness Most practical BFT SMR protocols today are designed for *partial synchrony*: they guarantee safety at all times but ensure liveness only during periods of sufficient synchrony. In practice, this is typically approximated using timeouts—when a timeout elapses, the protocol suspects a failure and triggers a new leader election to restore progress.

Unfortunately, many existing protocols degrade significantly when these timeouts occur. This is problematic because timeouts are notoriously hard to configure: too short, and they trigger false suspicions under benign delays; too long, and they delay recovery from genuine faults. As a result, timeout violations are common in real-world deployments, often caused by network delays, congestion, or simple replica failures. Contrary to the implicit assumption that such blips are rare, we argue they are intrinsic to the nature of partial synchrony. As such, protocols must perform well not only when synchorny holds, but also in the aftermath of its violation.

However, today's BFT protocols often fail to meet this standard. For instance, Hot-

Stuff [192] exhibits a pronounced performance degradation—or *hangover*—that persists even after a blip resolves. The root cause is that protocols like HotStuff and PBFT [30] tightly couple data dissemination with agreement: all operations must be both sequenced and broadcast through the designated leader. During a leader failure or asynchronous interval, this leads to backlogs. After recovery, these backlogs must be cleared, but progress is limited by the available excess throughput capacity, resulting in a sluggish recovery.

To support *seamless* recovery, Autobahn decouples data dissemination and agreement into two distinct layers. The first is a data layer that continues disseminating transactions at the pace of the network, even during periods of asynchrony. The second is a consensus layer that consumes the outputs of the data layer to achieve total order.

To enable seamless operation, the data layer must disseminate transactions in a manner that is: (*i*) *reliable*, ensuring that only data that actually exists is considered during agreement; (*ii*) *responsive*, progressing at the speed of the network; and (*iii*) *indexable*, to support fast and efficient recovery.

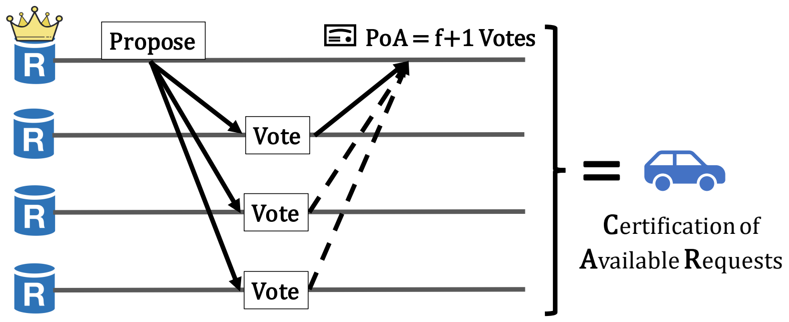

Figure 5.1: A car represents the process of certifying the availability of a request.

Cars and Lanes The key unit in Autobahn is the *car* (Fig. 5.1)—a conceptual structure that represents reliable data dissemination. To disseminate a transaction (or a batch of transactions), a replica must obtain a Proof of Availability (*PoA*): a quorum of acknowledgements confirming that at least one correct replica has durably stored the data.

Every replica in Autobahn continuously batches and disseminates transactions by constructing its own cars, independently of others. We say that each replica operates in its own *lane*, producing cars at its own rate (Fig. 5.2). This design maximizes throughput and allows Autobahn to reliably disseminate transactions—even in the presence of Byzantine failures or consensus interruptions. Within each lane, cars are chained together via hash references, forming an implicit order. Each car transitively proves the availability of all of its predecessors, enabling efficient indexing.

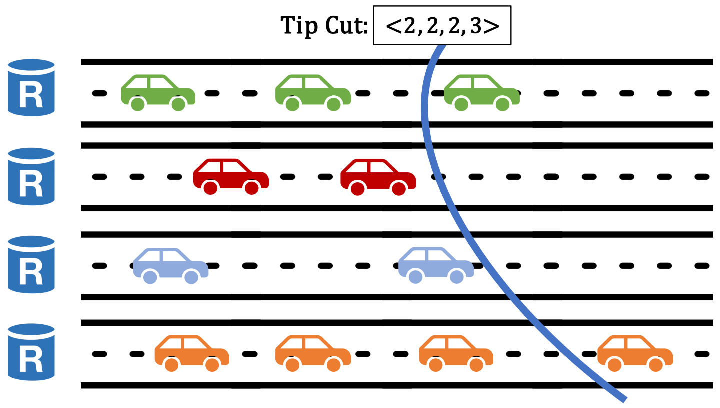

Figure 5.2: Autobahn chains cars into lanes, with each replica operating its own lane independently. To establish a total order, it suffices to reach agreement a cut—a set of lane tips—which implicitly captures a consistent snapshot of the entire system state.

Reaching Agreement While lanes ensure that transactions are reliably disseminated, they only provide weak consistency. For instance, different replicas may observe inconsistent views of the same lane, where neither may be a prefix of the other. The role of the consensus layer is to reconcile these views and impose a total order on the certified transactions.

Autobahn's lane structure helps streamline the process. In particular, it suffices to reach agreement on a *cut*—the latest car (or *tip*) in each lane (Fig. 5.2). Since each tip subsumes all prior cars in its lane, reaching agreement on a cut suffices to agree on and ordering for all underlying data.

Autobahn's architecture supports a flexible consensus backend. Systems can instantiate Autobahn with a consensus protocol of their choice, and "motorize" it to achieve high throughput and seamlessness. By default, Autobahn uses a latency-optimized variant of PBFT [30], as illustrated in Figure 5.3.

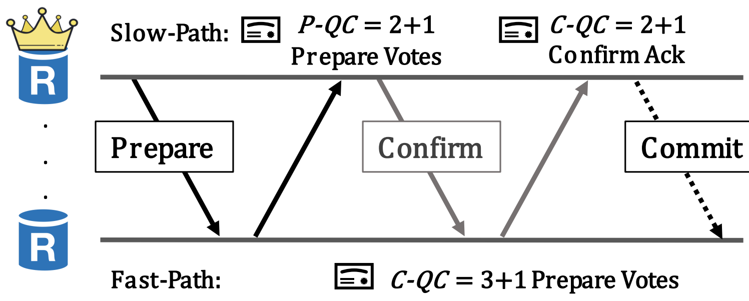

Figure 5.3: Autobahn's consensus layer follows a classic PBFT-based approach. To reach agreement, a leader replica collects a quorum of votes over two rounds. In fault-free settings, Autobahn can reach agreement within a single round (fast path).

Evaluation Autobahn offers best-in-class performance. In matches the throughput of modern DAG-based BFT protocols such as Bullshark [165], while cutting latency by more than half. At the same time, it matches HotStuff's latency without suffing from its recovery hangovers.

5.1.2 BeeGees: Stayin' Alive in Chained BFT

BeeGees [74] is a BFT SMR protocol designed to improve the liveness of *chained* BFT protocols such as HotStuff. Chained BFT protocols exploit the structural symmetry between consecutive consensus phases to pipeline agreement instances: for example, each voting round serves simultaneously as the *commit* phase of one instance, and the *pre-commit* phase of the next (Fig. 5.4). This pipelining amortize cryptographic overheads and reduces latency by minimize the wait-time between consecutive proposals. To further enhance fairness and resilience, many chained BFT protocols also employ proactive leader rotation—known as *leader-speaks-once* (LSO)—in which each leader makes only a single proposal before passing control to the next.

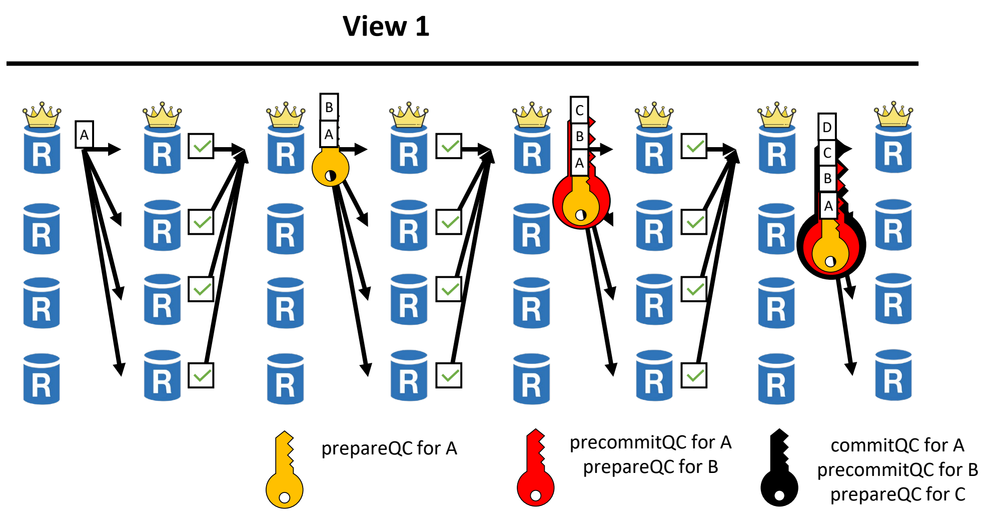

Figure 5.4: An illustration of chained HotStuff. Each quorum certificate (QC) simultaneously serves as the prepareQC, precommitQC, and commitQC for a different proposal.

Liveness Woes In practice, however, this combination of pipelining and leader rotation significantly weakens liveness under partial synchrony or failure. Traditional chained LSO protocols (CLSO) require multiple *consecutive* correct leaders to make progress—a fragile assumption that often fails in realistic deployments with Byzantine or slow nodes. HotStuff, for example, needs at least four consecutive correct leaders to commit a proposal. In the presence of faults (or even simple delays), this requirement can severely impact latency. With $n = 100$ replicas, we find that HotStuff takes an average of 12 rounds, representing a 4x increase over the ideal case.

BeeGees, in a Nutshell To address this limitation, BeeGees introduces a CLSO protocol that safely removes the need for consecutive correct leaders. The key insight is simple. While traditional BFT protocols disregard *prepare* votes during view changes—since they are not required for *safety*[1]—BeeGees leverages them to enhance *liveness*. Specifically, prepare votes help BeeGees distinguish between omission/asynchrony and active equivocation.

In the case of omission or asynchrony, BeeGees uses prepare votes to prevent the formation of “dangerous” quorum certificates (QCs)—those that might otherwise allow different replicas to commit conflicting values. In the case of equivocation, BeeGees detects when such a QC *could have* formed and proactively aborts the corresponding proposal. Furthermore, prepare votes allow BeeGees to detect *implicit* QCs: those that should have formed under a correct leader, but were witheld by a subsequent faulty one. Alltogether, these mechanisms allow BeeGees to safely commit proposals even without consecutive correct leaders.

Evaluation BeeGees refined commit rule significantly improves the protocol’s ability to "stay alive" in the face of faulty of slow leaders. In a 100-replica deployment, BeeGees reduces expected commit latency from 12 rounds (in HotStuff) to just 4.5 rounds—a 3x improvement. It also slashes worst-case latency under the most unfavorable leader schedules from 129 rounds (HotStuff) to 18—a 7x reduction.

Importantly, BeeGees retains the effieciency characteristics of modern BFT protocols. Under synchronous and failure-free conditions, it commits in just two phases—the best in class [67, 91], and theoretical minimum [2]. It requires only quadratic communication when using threshold signatures [80], and can be reduced to linear overhead with succinct cryptographic proofs such as SNARKs [3].

[1]A notable exception includes optimistic fast-path protocols like Zyzzyva [101] and SBFT [80], which use superquorums of all *n* prepare votes.

5.1.3 Morty: Scaling Concurrency Control with Re-Execution

Morty [28] is a high performance, crash fault tolerant (CFT) key-value store that overcomes contention bottlenecks in transactional workloads by leveraging excess CPU resources to perform dynamic transaction re-execution. This approach yields substantial performance gains, particularly in geo-distributed deployments which are especially prone to high abort rates under contention.

Performance Struggles Under Contention For ease of development, applications increasingly demand strong consistency guarantees—they rely on transactions for atomicity, and serializability for integrity. Enforcing these guarantees, however, imposes overhead, especially in workloads with high contention, where transactions must be carefully serialized. Traditional concurrency control mechanisms struggle in these scenarios: under two-phase locking (2PL), conflicting transactions block and may deadlock; under optimistic concurrency control (OCC), they abort and retry. The problem is exacerbated in geo-distributed environments, where long transaction durations increase the likelihood of overlapping execution. We formalize this intuition through the concept of *conflict windows*, which upper-bound the achievable performance of any transactional system under contention.

Scaling in the Face of Contention Existing concurrency control schemes fail to efficiently sequence conflict windows. For example, OCC systems often rely on randomized exponential backoff to reduce conflict likelihood, but this introduces unnecessary gaps between conflict windows, leaving performance untapped. Morty addresses this inefficiency through dynamic *re-execution*. Rather than relying solely on speculative execution and validation, Morty eagerly exposes the writes of in-progress transactions and notifies concurrent readers who may have missed those writes. In an effort to avoid aborts, clients altruistically roll back and re-execute from the point of the missed write.

This mechanism tightly aligns conflict windows and maximizes concurrency. Figure 5.5 illustrates a simple re-execution scenario with three concurrent transactions.

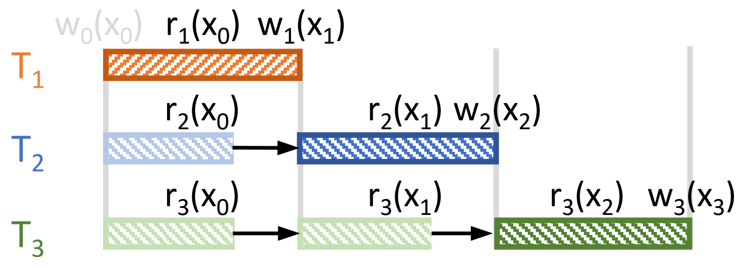

Figure 5.5: Transaction re-execution with three concurrent transactions. T_2 and T_3 initially miss the write of T_1, and re-execute once w_1 becomes visible. This repeates for T_3 upon missing the write of T_2.

To support re-execution, Morty adopts a continuation passing style (CPS) API for programming transactions—an approach popularized by systems such as FaRM [52]. This allows each transaction step to express both its current *context* (the state resulting from previous execution) and its *continuation* (what to execute next). Clients cache both the context and continuation, and dynamically materialize a new execution upon learning of a previously missed, relevant write.

While re-execution incurs overhead comparable to aboring and retrying—both clients and servers need to consume additional CPU resources—this cost is absorbed by otherwise idle resources. Under high contention, workloads are often bottlenecked by coordination delays (*e.g.*, long transaction backoff), not compute capacity. For instance, we find that both Google Spanner [40] and TAPIR [195] utilize only about 17% of a single core when operating under high contention. Morty, by contrast, harnesses these idle CPU resources to re-execute transactions rather than abort them. As a result, Morty commits over 99% of transactions, dramatically reducing wasted work.

Evaluation Our evaluation shows that Morty significantly outperforms existing transactional CFT data stores in both regional and and geo-distributed settings. For example, on TPC-C with 100 warehouses, Morty achieves a 7.4x throughput improvement over Spanner, and a 4.4x improvement over TAPIR. Under extreme contention, these gains grow to 95x and 52x, respectively.

5.2 Acknowledgements

I would like to acknowledge the contributions of my co-authors to the work presented in this dissertation.

Matthew Burke was instrumental in the development of the initial Basil prototype (Chapter 3) and contributed significantly to the experimental infrastructure. Yunhao Zhang and Zheng Wang helped implement the HotStuff and BFT-SMaRt baselines, respectively. Daniel Weber contributed much of the cryptographic infrastructure used across all systems.

Neil Giridharan made major contributions to the design and implementation of Pesto's database engine (Chapter 4). Liam Arzola and Roberto Toledo helped devlop the benchmark workloads. Along with Benton Li, Liam also built the CockroachDB baseline. Shir Cohen and Oliver Matte assisted with implementing the PostgreSQL baseline. Samyu Yagati and Suyash Gupta contributed to early designs of the Pesto protocol.

Finally, I would like to express my deep gratitude to Natacha Crooks and Lorenzo Alvisi for their profoundly important role in all the work presented in this dissertation.

This research was supported in part by the National Science Foundation under grants CSR-17620155, CNS-CORE 2008667, CNSCORE 2106954, as well as by the Initiative for Cryptocurrencies and Contracts (IC3).

BIBLIOGRAPHY

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[3] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. Malicious Security Comes For Free in Consensus with Leaders. *Cryptology ePrint Archive*, 2020.

[4] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[5] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. Blockchain Technology in Healthcare: A Systematic Review. In *Healthcare*, 2019.

[6] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[7] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. *arXiv preprint:1708.03778*, 2017.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2018.

[9] Suveen Angraal, Harlan M Krumholz, and Wade L Schulz. Blockchain Technology: Applications in Health Care. *Circulation: Cardiovascular Quality and Outcomes*, 2017.

[10] Jason Ansel and Marek Olszewski. BFTree–Scaling HotStuff to Millions of Validators. Celo Whitepaper. Accessed: 2025-07-15.

[11] ANSI X3.135-1992. American National Standard for Information Systems – Database Language – SQL, 1992.

[12] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[13] Diem Association. Diem. `https://www.diem.com/en-us/`. Accessed: 2025-07-15.

[14] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a SQL system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[15] European Central Bank. Digital Euro. `https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html`. Accessed: 2025-07-15.

[16] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State Machine Replication in the Libra Blockchain. Technical report, The Libra Association, Facebook Inc., 2019.

[17] Rida Bazzi and Maurice Herlihy. Clairvoyant State Machine Replication. *Information and Computation*, 2022.

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995.

[19] Daniel Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. Ed25519: high-speed high-security signatures. `http://ed25519.cr.yp.to/`. Accessed: 2025-07-15.

[20] Philip A Bernstein and Nathan Goodman. Multiversion Concurrency Control–Theory and Algorithms. *ACM Transactions on Database Systems (TODS)*, 1983.

[21] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Reading, 1987.

[22] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal Aspects of

Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 1979.

[23] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2014.

[24] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *International Conference on Practice and Theory in Public Key Cryptography (PKC)*, 2003.

[25] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-Signatures for Smaller Blockchains. In *Proceedings of the International Conference on The Theory and Application of Cryptology and Information Security (Asiacrypt)*, 2018.

[26] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2003.

[27] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Masters thesis, University of Guelph, 2016.

[28] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2023.

[29] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography (Extended Abstract). In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[30] Miguel Castro, Barbara Liskov, et al. Practical Byzantine Fault Tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[31] T-H Hubert Chan, Rafael Pass, and Elaine Shi. PaLa: A Simple Partially Synchronous Blockchain. *Cryptology ePrint Archive*, 2018.

[32] T. Chandra, R. Griesmer, and J. Redstone. Paxos Made Live – An Engineering

Perspective. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[33] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)*, 1996.

[34] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. UpRight Cluster Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[35] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[36] CloudLab Team. CloudLab. `https://www.cloudlab.us`. Accessed: 2025-07-15.

[37] Cockroach Labs. CockroachDB. `https://www.cockroachlabs.com/`. Accessed: 2025-07-15.

[38] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. Secret Key Recovery in a Global-Scale End-to-End Encryption System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[39] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[40] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[41] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[42] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[43] Wei Dai and collaborators. Crypto++. `https://github.com/weidai11/cryptopp/`. Accessed: 2025-07-15.

[44] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

[45] Boneh Dan, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. In *Proceedings of the International Conference on The Theory and Application of Cryptology and Information Security (Asiacrypt)*, 2001.

[46] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based Mempool and Efficient BFT Consensus. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[47] Deloitte. Continuous interconnected supply chain. `https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/technology/lu-blockchain-internet-things-supply-chain-traceability.pdf`. Accessed: 2025-07-15.

[48] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2013.

[49] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding Concurrency to Smart Contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.

[50] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2013.

[51] Tobias Distler and Rüdiger Kapitza. Increasing Performance in Byzantine Fault-

Tolerant Systems with On-Demand Replica Consistency. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[52] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[53] Sisi Duan, Sean Peisert, and Karl N Levitt. *h*BFT: Speculative Byzantine Fault Tolerance with Minimum Cost. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2014.

[54] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 1988.

[55] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A Shared Database on Blockchains. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2019.

[56] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 1976.

[57] Jose M Faleiro and Daniel J Abadi. Rethinking Serializable Multiversion Concurrency Control (Extended Version). *arXiv preprint:1412.2324*, 2014.

[58] State Farm. State Farm and USAA Work Together to Test a Blockchain Solution. `https://newsroom.statefarm.com/blockchain-solution-test-for-subrogation/`, 2019. Accessed: 2025-07-15.

[59] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 1985.

[60] Aptos Foundation. Aptos. `https://aptosfoundation.org/`. Accessed: 2025-07-15.

[61] Aptos Foundation. Move–A Web3 Language and Runtime. `https://aptos.dev/en/network/blockchain/move`. Accessed: 2025-07-15.

[62] Ethereum Foundation. Ethereum. `https://ethereum.org/en/`. Accessed: 2025-07-15.

[63] Ethereum Foundation. Ethereum Virtual Machine. `https://ethereum.org/en/developers/docs/evm/`. Accessed: 2025-07-15.

[64] Solana Foundation. Solana. `https://solana.com/`. Accessed: 2025-07-15.

[65] Sui Foundation. Sui. `https://sui.io/`. Accessed: 2025-07-15.

[66] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient Middleware for Byzantine Fault Tolerant Database Replication. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[67] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2022.

[68] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023.

[69] Craig Gentry and Zulfikar Ramzan. Identity-Based Aggregate Signatures. In *International Workshop on Public Key Cryptography*, 2006.

[70] GigaSpaces. Amazon Found Every 100ms of Latency Cost them 1% in Sales. `https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/`, 2023. Accessed: 2025-07-15.

[71] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[72] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. No-Commit Proofs: Defeating Livelock in BFT. *Cryptology ePrint Archive*, 2021.

[73] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[74] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. BeeGees: Stayin' Alive in Chained BFT. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2023.

[75] Google. Protobuf. `https://protobuf.dev/`. Accessed: 2025-07-15.

[76] Carnegie Mellon Database Group. Benchbase. `https://db.cs.cmu.edu/projects/benchbase/`. Accessed: 2025-07-15.

[77] Carnegie Mellon Database Group. Peloton–The Self-Driving Database Management System. `https://db.cs.cmu.edu/peloton/`. Accessed: 2025-07-15.

[78] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org`. Accessed: 2025-07-15.

[79] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2010.

[80] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a Scalable and Decentralized Trust Infrastructure. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2019.

[81] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable Virtual Machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[82] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[83] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL Database. In *In Proceedings of the IEEE International Conference on Pervasive Computing and Applications (ICPCA)*, 2011.

[84] James Hendricks, Shafeeq Sinnamohideen, Gregory R Ganger, and Michael K Reiter. Zzyzx: Scalable Fault Tolerance through Byzantine Locking. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2010.

[85] Maurice Herlihy and Mark Moir. Enhancing Accountability and Trust in Distributed Ledgers. *arXiv preprint:1606.07490*, 2016.

[86] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: a Raft-based HTAP Database. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2020.

[87] IBM. Blockchain World Wire (Deprecated). `https://www.ibm.com/support/pages/ibm-blockchain-world-wire-revolutionize-cross-border-payments`, 2020. Accessed: 2025-07-15.

[88] MongoDB Inc. MongoDB. `https://www.mongodb.com/`. Accessed: 2025-07-15.

[89] Yugabyte Inc. YugabyteDB. `https://www.yugabyte.com/`. Accessed: 2025-07-15.

[90] Kazuharu Itakura and Katsuhiro Nakamura. A Public-Key Cryptosystem Suitable for Digital Multisignatures. *NEC Research & Development*, 1983.

[91] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-HotStuff: A Fast and Robust BFT Protocol for Blockchains. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2023.

[92] JR Jordan, J Banerjee, and RB Batman. Precision Locks. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1981.

[93] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2011.

[94] Jovan Kalajdjieski, Mayank Raikwar, Nino Arsov, Goran Velinov, and Danilo Gligoroski. Databases Fit for Blockchain Technology: A Complete Overview. *Blockchain: Research and Applications*, 2023.

[95] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[96] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2023.

[97] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-Fairness for Byzantine Consensus. In *Proceedings of Advances in Cryptology (CRYPTO)*, 2020.

[98] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 583–598, 2018.

[99] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 2000.

[100] R. Kotla and M. Dahlin. High Throughput Byzantine Fault Tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2004.

[101] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[102] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[103] S Krishnapriya and Greeshma Sarath. Securing Land Registration using Blockchain. In *Proceedings of the International Conference on Computing and Network Communications (CoCoNet)*, 2020.

[104] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 1981.

[105] Klaus Kursawe. Wendy, the Good Little Fairness Widget–Achieving Order Fairness for Blockchains. In *Proceedings of the ACM Conference on Advances in Financial Technologies (AFT)*, 2020.

[106] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 2001.

[107] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[108] Leslie Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2005.

[109] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[110] Leslie Lamport. Byzantizing Paxos by Refinement. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2011.

[111] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

[112] Butler Lampson. The ABCD's of Paxos. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[113] Costin Leau. Retwis-J. `https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`. Accessed: 2025-07-15.

[114] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proceedings of the European Dependable Computing Conference (EDCC)*, 2016.

[115] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[116] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The Paxos Register. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2007.

[117] Greg Linden. Google VP Marissa Mayer at Web 2.0. `https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`, 2006. Accessed: 2025-07-15.

[118] Linux Foundation. Change Healthcare Case Study. `https://www.lfdecentralizedtrust.org/case-studies/change-healthcare-case-study`. Accessed: 2025-07-15.

[119] Linux Foundation. Hyperledger Fabric. `https://www.lfdecentralizedtrust.org/projects/fabric`. Accessed: 2025-07-15.

[120] Barbara Liskov. From Viewstamped Replication to Byzantine Fault Tolerance. In *Replication: Theory and Practice*. Springer, 2010.

[121] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, Massachusetts Institute of Technology, 2012.

[122] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine Faulty Clients in a Quorum System. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[123] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. Byzantine Fault-Tolerant Transaction Processing for Replicated Databases. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*, 2011.

[124] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible Byzantine Fault Tolerance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[125] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Distributed Computing*, 1998.

[126] J-P Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2006.

[127] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. BigchainDB: A Scalable Blockchain Database. BigchainDB-Whitepaper, 2016.

[128] Ralph C Merkle. A Digital Signature based on a Conventional Encryption Function. In *Proceedings of Advances in Cryptology (CRYPTO)*, 1987.

[129] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-Subgroup Multisignatures. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2001.

[130] Microsoft. Azure Confidential Ledger. `https://azure.microsoft.`

com/en-gb/products/azure-confidential-ledger/#overview. Accessed: 2025-07-15.

[131] Microsoft. Microsoft Confidential Consortium Framework. https://www.microsoft.com/en-us/research/project/confidential-consortium-framework/. Accessed: 2025-07-15.

[132] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[133] Priti Mishra and Margaret H Eich. Join Processing in Relational Databases. *ACM Computing Surveys (CSUR)*, 1992.

[134] Andrew Moon. ed25519-donna. https://github.com/floodyberry/ed25519-donna. Accessed: 2025-07-15.

[135] Iulian Moraru, David G Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[136] J.P. Morgan. Kinexys. https://www.jpmorgan.com/kinexys/index. Accessed: 2025-07-15.

[137] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[138] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[139] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *arXiv preprint:1903.01919*, 2019.

[140] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2015.

[141] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[142] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[143] Oracle. MySQL. `https://www.mysql.com/`. Accessed: 2025-07-15.

[144] Oracle. Oracle. `https://www.oracle.com/database/index.html`. Accessed: 2025-07-15.

[145] Ricardo Padilha, Enrique Fynn, Robert Soulé, and Fernando Pedone. Callinicos: Robust Transactional Storage for Distributed Data Structures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[146] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and Robust Storage for Cloud Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[147] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[148] Seo Jin Park and John Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[149] Andrew Pavlo. What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[150] Fernando Pedone and Nicolas Schiper. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society*, 2012.

[151] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. FalconDB: Blockchain-based Collaborative Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2020.

[152] Huma Pervez, Muhammad Muneeb, Muhammad Usama Irfan, and Irfan Ul Haq. A comparative analysis of dag-based blockchain architectures. In *Proceedings*

of the IEEE International Conference on Open Source Systems and Technologies (ICOSST), 2018.

[153] PingCAP. TiDB. `https://www.pingcap.com/`. Accessed: 2025-07-15.

[154] Miguel Pires, Srivatsan Ravi, and Rodrigo Rodrigues. Generalized Paxos Made Byzantine (and Less Complex). *Algorithms*, 2018.

[155] Serguei Popov and Q Lu. IOTA: Feeless and Free. *IEEE Blockchain Technical Briefs*, 2019.

[156] David P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems (TOCS)*, 1983.

[157] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and Probabilistic Leaderless BFT Consensus through Metastability. *arXiv preprint:1906.08936*, 2019.

[158] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 1978.

[159] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. Distributed databases: Dichotomy and Fusion. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2021.

[160] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990.

[161] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. chainifyDB: How to get rid of your Blockchain and use your DBMS instead. In *In Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[162] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[163] Victor Shoup. Practical threshold signatures. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2000.

[164] Avid Silbershatz, Henry Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 6th edition, 2010.

[165] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2022.

[166] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-Throughput BFT for Blockchains. *arXiv preprint:1906.05552*, 2019.

[167] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[168] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-Phase BFT with Linearity. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2022.

[169] Xiao Sui, Qichang Liu, Sisi Duan, and Haibin Zhang. Pike: Two-Phase BFT with Linearity and Flexible View Change. *IEEE Transactions on Computers*, 2025.

[170] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID transactions–SOSP'21 Artifact. `https://github.com/fsuri/Basil_SOSP21_artifact`. Accessed: 2025-07-15.

[171] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[172] Florian Suri-Payer, Neil Giridharan, Liam Arzola, Shir Cohen, Lorenzo Alvisi, and Natacha Crooks. Pesto: Cooking up High Performance BFT Queries. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2025.

[173] Pierre Sutra and Marc Shapiro. Fast Genuine Generalized Consensus. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2011.

[174] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-Scalable

Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[175] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2020.

[176] BFT-SMaRt Team. Byzantine Fault-Tolerant (BFT) State Machine Replication (SMaRt) v1.2. `https://github.com/bft-smart/library`. Accessed: 2025-07-15.

[177] BLAKE3 Team. BLAKE3. `https://github.com/BLAKE3-team/BLAKE3`. Accessed: 2025-07-15.

[178] CometBFT Team. CometBFT. `https://github.com/cometbft/cometbft`. Accessed: 2025-07-15.

[179] Cypherium Team. Cypherium. `https://www.cypherium.io/`. Accessed: 2025-07-15.

[180] Kwil Team. Kwil. `https://www.kwil.com/`. Accessed: 2025-07-15.

[181] Meter Team. Meter. `https://www.meter.io/`. Accessed: 2025-07-15.

[182] Transaction Processing Performance Council. TPC-C Benchmark. `http://www.tpc.org/tpcc`. Accessed: 2025-07-15.

[183] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[184] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2014.

[185] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[186] Williot. IBM Food Trust: Transforming the Future of Food Safety and Supply Chains. `https://www.wiliot.com/podcasts/ibm-food-trust-transforming-the-future-of-food-safety-and-supply-chains`, 2023. Accessed: 2025-07-15.

[187] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[188] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2018.

[189] Mihalis Yannakakis. Serializability by Locking. *Journal of the ACM (JACM)*, 1984.

[190] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[191] Maofan Yin. libhotstuff. `https://github.com/hot-stuff/libhotstuff`. Accessed: 2025-07-15.

[192] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[193] Clement T. Yu and CC Chang. Distributed Query processing. *ACM Computing Surveys (CSUR)*, 1984.

[194] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling Blockchain via Full Sharding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[195] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[196] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. When Is Operation Ordering Required in Replicated Transactional Storage? *IEEE Data Engineering Bulletin*, 2016.

[197] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine Ordered Consensus without Byzantine Oligarchy. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[198] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An Overview on Smart Contracts: Challenges, Advances and Platforms. *Future Generation Computer Systems*, 2020.