Finance and Economics Discussion Series

Federal Reserve Board, Washington, D.C.
ISSN 1936-2854 (Print)
ISSN 2767-3898 (Online)

Heraclius: A Byzantine Fault Tolerant Database System with Potential for Modern Payments Systems

James Lovejoy, Tarakaram Gollamudi, Jeremy Kassis, Narayanan Pillai, Jeremy Brotherton, and Eric Thompson

2025-012

Please cite this paper as:
Lovejoy, James, Tarakaram Gollamudi, Jeremy Kassis, Narayanan Pillai, Jeremy Brotherton, and Eric Thompson (2025). "Heraclius: A Byzantine Fault Tolerant Database System with Potential for Modern Payments Systems ," Finance and Economics Discussion Series 2025-012. Washington: Board of Governors of the Federal Reserve System, https://doi.org/10.17016/FEDS.2025.012.

Heraclius: A Byzantine Fault Tolerant Database System with Potential for Modern Payments Systems

James Lovejoy, FRB Boston; Tarakaram Gollamudi, FRB Boston; Jeremy Kassis, FRB San Francisco; Narayanan Pillai, FRB San Francisco; Jeremy Brotherton, FRB NIT; and Eric Thompson, FRB Boston

Abstract

Modern payments systems are critical infrastructure for the US and global economy, and they all utilize computing systems to facilitate transactions. These computing systems can be vulnerable to failures and an outage of a payment system could cause a serious ripple effect throughout the economy it supports.

Commonly used designs in existing distributed computer systems often lack a built-in defense against certain types of failures (e.g., malicious attacks and silent data corruption) and rely on preventing these failures from happening in the first place via techniques external to the system itself. These computer system failures can cause downtime in the systems (e.g., modern payments systems) that rely on them. Byzantine Fault Tolerant (BFT) systems have the potential of improved resiliency and security. BFT systems can tolerate a larger range of failure modes than contemporary designs but suffer from performance challenges. Our work sought to design and evaluate a scalable BFT architecture and compare its properties to other database architectures used in payments infrastructure. This analysis is intended to better understand technical tradeoffs and is agnostic to broader policy or operational considerations.

In this paper, we present Heraclius, a parallelizable leader-based, BFT key-value store that could be extended for use in payment systems. Heraclius executes transactions in parallel to achieve high transaction volumes. We analyze the scalability of the protocol, bottlenecks and potential solutions to the bottlenecks. We ran the prototype implementation with up to 256 nodes and achieved a transactional volume of 110 thousand operations per second with a transaction latency 0.2 seconds.

Introduction

Payment systems are expected to provide speed, be inexpensive, provide high levels of availability and correctness of computation, and operate in an adversarial environment. Downtime in such a system would render it unable to settle payments, and the economy could experience significant disruption. Many payment systems leverage distributed computing to provide Crash Fault Tolerance (CFT), which is relatively good at delivering speed and low cost but must rely on walled gardens and other protections in order to provide high availability while operating in an adversarial environment. In contrast, Byzantine Fault Tolerance (BFT) techniques could potentially be used to develop systems with high availability and guarantee correctness in the presence of malicious parties and silent data corruption.[i] However, BFT systems typically offer worse performance than contemporary CFT designs and may have novel security or operational drawbacks. In this paper we present Heraclius, a novel BFT system that leverages

parallelism to achieve greater performance while tolerating byzantine faults. Such a system could have interesting applications in the payments system landscape.

Fault Tolerance and Electronic Payments

Electronic payment systems, like FedNow, the Automated Clearing House (ACH), Venmo, and Visa, fundamentally execute transfers of a given amount of funds from the payer's account balance to the payee's. These traditional electronic payment systems often rely on a database management system (DBMS) as a ledger to record transactions and user balances. The DBMS often runs as a distinct application from the payment system.

A fault is an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure, and electronic payments systems suffer from hardware failures and their software may contain bugs.[ii] Many of these faults are expected to occur with some regularity due to inherent limitations of hardware reliability, software correctness, and external events like natural disasters. Broadly speaking, failures in hardware or software components can result in two classes of faults: crash faults and byzantine faults. A crash fault is a type of failure where the component simply stops processing and no longer responds to requests (e.g., due to power loss). For a byzantine fault, the component could exhibit a crash fault, but also process data incorrectly or respond selectively to requests. Any behavior that is not correct for the component in question is considered byzantine. Byzantine faults can be caused by a larger number of underlying failures than crash faults, from software bugs to cosmic rays or malicious attacks. A system that is tolerant to crash faults is called crash-fault tolerant (CFT), and a system tolerant to byzantine faults is called byzantine-fault tolerant (BFT).

A byzantine fault in a CFT system can cause violations of the data integrity, cause downtime, and might be difficult to detect. In CFT systems, the risks associated with byzantine faults are usually mitigated outside the system, usually through careful code review, backups, intrusion detection, and physical security. These mitigation strategies can be heavily affected by human factors and can be costly to implement. An alternative BFT design could mitigate many of these risks automatically and predictably in software, allowing the system operator to reduce costs associated with existing mitigation strategies, or simply to provide additional assurance through defense-in-depth. In a world where electronic payment systems are critical national infrastructure and faults cannot be tolerated, BFT could be a desirable property. However, BFT systems typically have throughput scalability limitations which might make them inappropriate for systems that require high throughput capacity.

Consensus Algorithms

The DBMS used to implement contemporary payment systems are often architected as distributed systems. This includes most, if not all, traditional electronic payment rails. As a simple example, a payment rail having a back-up location for disaster recovery purposes would be considered a distributed system.

Distributed systems leverage consensus algorithms to provide fault tolerance, high availability and scalability. In distributed consensus, multiple instances of the service run on distinct servers (called "nodes") and communicate over a network. Each node performs the same operations in lockstep and votes to agree on the result (often called "state"). A group of nodes working together to perform distributed consensus for a particular application is called a "cluster." Distributed consensus provides fault tolerance by assuming that faults are uncorrelated, so that a fault in one node will happen

independently of the others. Consensus protocols rely on nodes performing redundant work to tolerate faults. Individual nodes perform the work, share their results, and vote for an outcome. Other nodes can require cryptographic attestation of voting decisions.

If a certain threshold of the total number of nodes, called a "quorum," continues to process operations and vote on the same result, the overall cluster tolerates faulty nodes by detecting invalid work and ignoring their votes. When a quorum of nodes in a cluster successfully votes on a new proposed state, a "decision" is made. Depending on the specific distributed consensus algorithm, the cluster can tolerate different classes of faults, crash or byzantine.

Since a quorum of nodes are required to vote on a decision, the throughput (i.e., number of transactions per second) of a cluster is limited by the slowest nodes. This makes systems based on a single cluster difficult to scale (i.e., increasing nodes increases throughput). Instead, scalable systems use additional techniques, in particular partitioning, where multiple distinct clusters work in parallel to divide up the work. This allows the overall system to scale by adding more clusters while the throughput of the individual clusters remains constant.

Prior Work

The following section identifies prior work in the areas of DBMS and BFT technology. These are presented for their technological contributions to the field.

Project Hamilton and PArSEC

Project Hamilton[iii] and PArSEC[iv] are specialized systems that leverage distributed consensus through the Raft[v] consensus algorithm and partitioning to implement a scalable electronic payment system that is crash fault tolerant. Project Hamilton demonstrated transaction throughput scaling to at least 1.8 million transactions per second with latency under 5 seconds, while PArSEC demonstrated throughput of 118 thousand transactions per second with linear scalability and similar latency. Both systems execute payments in parallel, resulting in increased throughput. They also demonstrate crash fault tolerance. However, neither system tolerates byzantine faults.

Project Hamilton can be augmented to provide detection of byzantine faults. This augmented system can detect that a byzantine fault has occurred and prevent further processing of transactions until the fault can be identified and resolved.[vi] Even though the system can prevent a byzantine fault from corrupting data, the system cannot safely remain available in the presence of byzantine faults. Hence a byzantine fault can affect the availability of the system.

Hotstuff and BFT-SMaRt

Hotstuff[vii] and BFT-SMaRt[viii] are leader-based BFT consensus algorithms for the permissioned setting, where the set of nodes is tightly controlled by the system operator. Relative to earlier BFT algorithms[ix], Hotstuff and BFT-SMaRt can support a much larger number of nodes participating in consensus without a degradation in throughput. While these BFT consensus algorithms provide high resiliency and better security, throughput, and latency, they are still single-cluster algorithms. As a result, they do not provide throughput scalability without additions to support adding more clusters working in parallel.

Bitcoin

The largest and longest running electronic payment system that tolerates byzantine faults today is Bitcoin. Bitcoin provides a very high degree of resiliency and security through its consensus mechanism. However, this comes at the expense of two challenges.

First, Bitcoin requires a large amount of energy consumption and compute power due to its permissionless proof-of-work consensus algorithm (Bitcoin nodes can join and leave at will and can be operated by anyone). This is not an essential component for a centralized payment system which operates in a permissioned setting, and a central bank could leverage other BFT algorithms which avoid the large costs.

Second, Bitcoin is a single cluster system that does not support parallelization or partitioning. Both factors cause Bitcoin to have high latency and low throughput. Today, Bitcoin can only accommodate approximately 7 transactions per second with latency ranging from 10 minutes to one hour, depending on the probability threshold the payee requires to consider a decision containing a payment final.[x]

Our Contributions

Our current work is an attempt to build a database suitable for an electronic payment system that is both BFT and horizontally scalable. We present Heraclius[xi] a scalable, distributed key-value store DBMS that leverages consensus protocols with cryptographic proofs to provide byzantine fault tolerance. Our novel inter-BFT cluster communication protocol allows composing multiple BFT clusters into a two layered network architecture to coordinate client requests and operate on data separately. The protocol retains byzantine fault tolerance for the overall system by proving authenticity and integrity of cross-cluster requests and responses. This unlocks the ability to partition data and process requests in parallel which increases the throughput by increasing the number of clusters working together.

We implemented and deployed a prototype version of the system in the cloud and quantitatively evaluated its scalability, throughput, and latency through benchmarking. We show that while we can build a secure protocol, retaining BFT for the overall system, the protocol design limits the scalability of the system compared to our prior work and has additional cost and complexity.

The remainder of this paper first describes our architecture, protocol design and security argument. Second, we describe our implementation and benchmarking setup. Third, we present our scalability benchmarking results. Finally, we compare our quantitative results to existing systems, provide a qualitative analysis of the tradeoffs associated with our design, and offer ideas for future work.

System Overview

Heraclius is a single key operation key-value database together with a distributed consensus algorithm. A key-value store is a type of database that stores data in simple pairs: a unique key and its corresponding value. The system consists of two distinct subsystems, the shard subsystem and coordinator subsystem. Each subsystem is a network of clusters. Each cluster contains multiple replicas of the service run on distinct servers (called “nodes”) and communicate over a network. A node (or replica) considered faulty if it deviates from the protocol specification.

The diagram below is a schematic representation of Heraclius with the shard subsystem, and the coordinator subsystem.

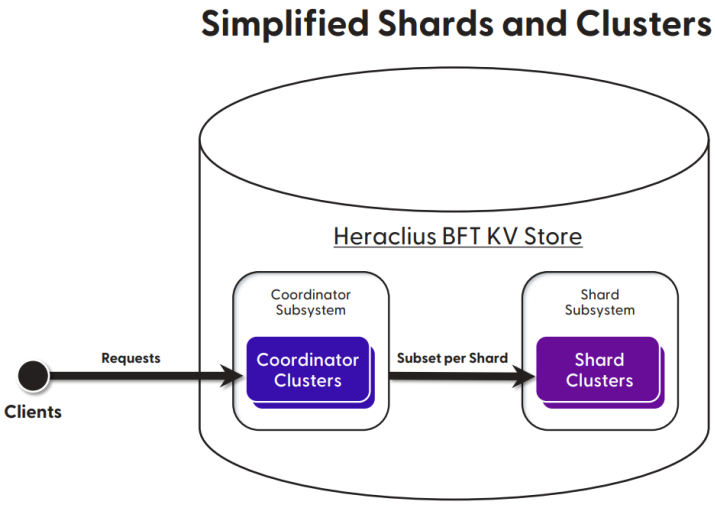


Figure 1: Heraclius components

BFT round: Each cluster has a designated leader which interacts with clients. The leader combines multiple requests from external clients or clusters and puts them into a proposal for voting. All nodes in the cluster process the requests, generate any responses and append them to the proposal, and vote whether to accept or reject the proposal. The leader aggregates the votes into a Quorum Certificate (QC) containing cryptographic signatures from each of the nodes providing a vote. A quorum certificate requires a super majority of the total nodes in the cluster agree on the same the proposal, after which the proposal becomes a decision and is final. Once the quorum decides, the leader dispatches requests with cryptographic attestations to decision queues of counterpart clusters. Counterpart clusters validate the decisions before extracting and processing any requests or responses from a remote cluster. The clusters structure the proposal so that a counterpart cluster can extract a batch containing only the requests and responses relevant to them from the overall proposal. At the end of each BFT round, the nodes extract the cluster-specific batches from the overall decision into "compact decisions". The votes in a QC sign a cryptographic proof that the compact decisions are subset of the original decision the cluster agreed upon and that the leader did not alter the requests or responses after the cluster voted on them. This validation is necessary to achieve BFT for the overall system where multiple distinct clusters are working in parallel. To tolerate f faults per cluster, we need to have $3f + 1$ nodes in each cluster and to decide, a quorum of $2f + 1$ non-faulty nodes is necessary (for further information, refer to *The Byzantine General's Problem* by Lamport et al.[xii]).

Coordinator clusters are responsible for handling external requests and distributing the requests amongst the shards. A coordinator cluster batches multiple requests from clients destined for different shard clusters into a proposal. Coordinators subscribe to compact decisions from shards, and forward shard responses back to external clients.

Shard clusters contain a partition of data are responsible for handling data operations. The shard clusters replicate compact decisions from coordinator clusters' decision queues, extract requests and generate responses. Clients validate coordinator QCs, coordinators validate shard QCs, and shards validate coordinator QCs.

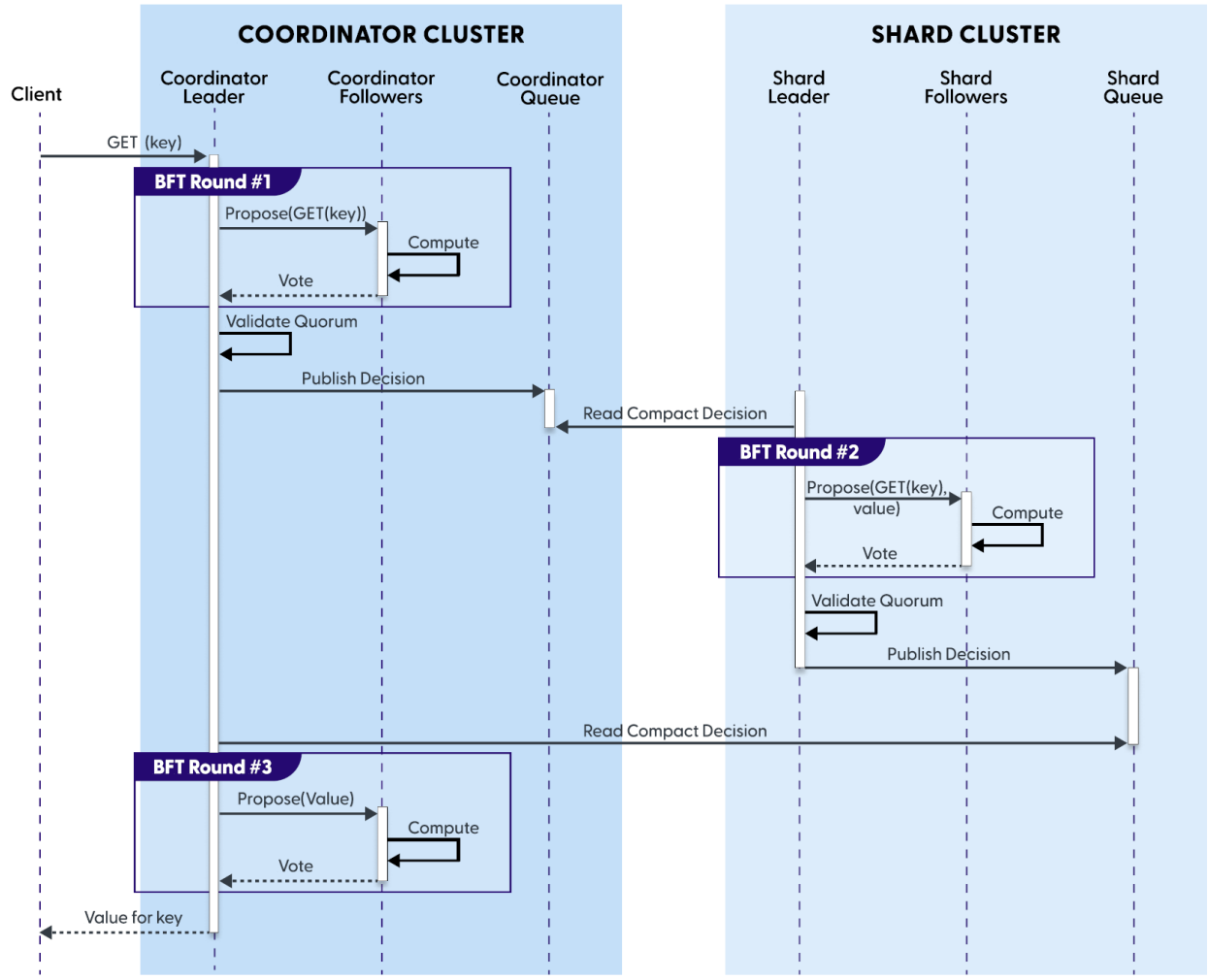
The slowest nodes of a group limit the throughput of a cluster because a quorum of nodes is required to vote on a decision. This makes systems based on a single cluster difficult to scale. Instead, in Heraclius, multiple coordinator and shard clusters work in parallel to divide up the work. This allows the overall system to scale by adding more clusters while the throughput of the individual clusters remains constant.

As an example, consider the case where a client requests a value corresponding to a key: GET (Key). This operation is analogous to querying an account balance. The client sends the request to a coordinator cluster, this choice of cluster could be arbitrary. The designated leader of the cluster receives this request. The leader creates a proposal with the GET(Key) request from the client, a corresponding GET(Key) request for the appropriate shard and sends it to other nodes for voting. Each node in the coordinator cluster re-generates the proposal and checks to make sure it matches the leader's proposal. If a supermajority of the nodes agrees on the proposal, then the leader publishes the decision to the decision queue. If a supermajority of nodes cannot agree on a proposal, a new leader is selected until agreement can be reached. This prevents faulty nodes from causing a fault in the overall cluster because a supermajority is required to decide and process any requests or responses.

The corresponding shard leader replicates compact decisions from the coordinator decision queue, validates the compact decision, extracts the GET(Key) request, looks up the value locally and appends a response to the coordinator with the value to the proposal. Each regenerates the proposal and compares it to the version proposed by the leader. When a supermajority of nodes agrees on the proposal, the leader publishes the decision to its decision queue.

The corresponding coordinator cluster leader reads the compact decision from the shard decision queue and the cluster nodes performs another BFT round. This is necessary to verify the shard's decision, extract the response from the shard, and append a response for the client to the proposal. Once a supermajority of nodes agrees on the proposal, the leader publishes the decision, and the client can validate the compact decision and extract the response.

The following figure depicts the timing sequence of the three BFT rounds that are required to complete the transaction.



Sequence Diagram of One Transaction

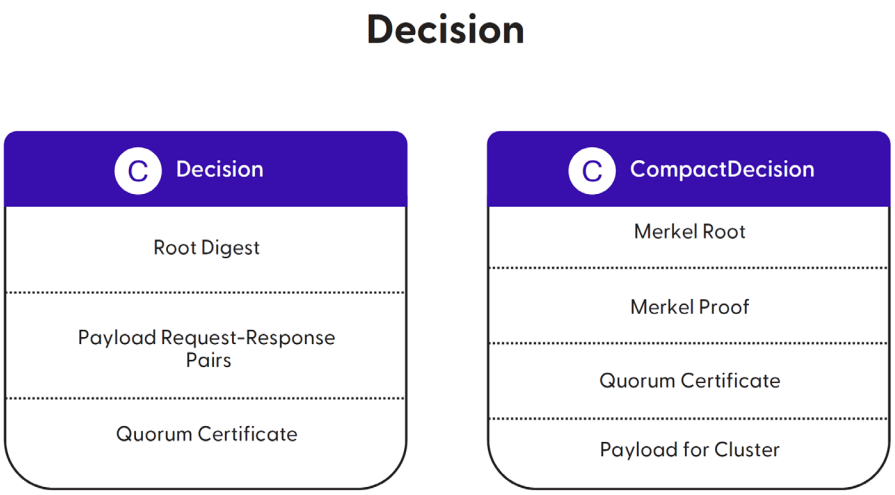
Cryptographic Proofs

Since the leader node manages the communication between cluster, the leaders could alter the compact decisions sent to remote clusters, inject faulty data through inter-cluster communication. Therefore, to maintain BFT guarantees the compact decisions must contain a proof that they are a valid subset of the original decision signed by a QC, and clusters must validate the proof before processing any requests or responses contained inside. A naive way to construct the compact decision is to provide the original decision that contains requests and responses for all n clusters. Each cluster must maintain this record for all n downstream or upstream clusters. All pairs of shards and coordinators, therefore, must exchange n^2 records or messages. This results in a significant increase in the volume of message exchanges as the number of clusters increases, which would negatively impact the scalability of the overall system.

Structuring the decisions as a Merkle tree reduces this communication overhead from n^2 to $n \log_2(n)$. The size of a Merkle proof that a particular set of requests and responses are included in the overall

decision is then log2(n). Since there are n of these proofs per cluster, exchanging compact decisions has a message overhead of $n \log_2(n)$, thus providing a significant reduction in the message volume.

As depicted in the figure below, a cluster decision contains the Merkle tree root digest, payload containing the requests and responses grouped by remote cluster, and a QC. In contrast, compact decisions for a particular cluster contain the Merkle tree root digest, Merkle proof, QC and payload containing only the requests and responses relevant to the cluster.



Data structure of decisions and compact decisions

Security argument

Our system retains byzantine fault tolerance, enforcing correctness and liveness, assuming every cluster has a quorum of non-faulty nodes available. An adversary with network-level access, or individual faulty or compromised nodes would be unable to generate a valid compact decision and convince other BFT clusters to process invalid requests or responses. The inter-cluster communication algorithm of replicating and validating remote compact decisions for all requests and response processing prevents injection of faulty data across BFT cluster boundaries. Our system's security thus depends on avoiding correlated failures within clusters, as any failure of an entire cluster would result in a breakdown of liveness or correctness for the whole system, even if other clusters are unaffected by faults themselves. Avoiding correlated failures could be difficult if node implementations or their deployment strategy are homogenous.

Diversity of Implementation and Deployment

To comprehensively address the challenges of node homogeneity, a deployed version of the system would have to leverage diversity of implementation, deployment, and control. A shared codebase can create systemic weaknesses in identical nodes, and a single bug can lead to simultaneous, shared vulnerability. Reliance on homogeneous hardware, operating systems, programming languages, and libraries across all system nodes, presents an attack and failure surface that could lead to catastrophic system collapse. The recent catastrophic global outages related to zero-day failures of Windows machines running the CrowdStrike[xiii] is a specific example. A system built from diverse implementations of the node software delivers a reliability advantage against Byzantine faults.

Natural disasters, infrastructure failures (black-outs, brown-outs), cyber-attacks, and physical attacks can disrupt the availability of co-located or proximally located systems. Diversity of Deployment refers to the intent that operators of the system shall deploy their nodes in more than one datacenter across multiple physical regions. Similarly, if a small set of individuals control access to deployment processes, a single process failure could cause faults in many the nodes. Diversity of control distributes sensitive access and deployment permissions between many individuals so that risks are not concentrated.

While diversity of control and a centrally controlled database may seem contradictory, it need not be. Even when a database system is centrally controlled, it can have multiple nodes, each of which is controlled by a different team with a different implementation. These teams can each design different approaches to maintain and operate their node. The result being that all nodes are under the domain of control of one organization, but each is deployed differently by different actors in different locations. This has the effect of de-correlating node failures while retaining overall control of the system.

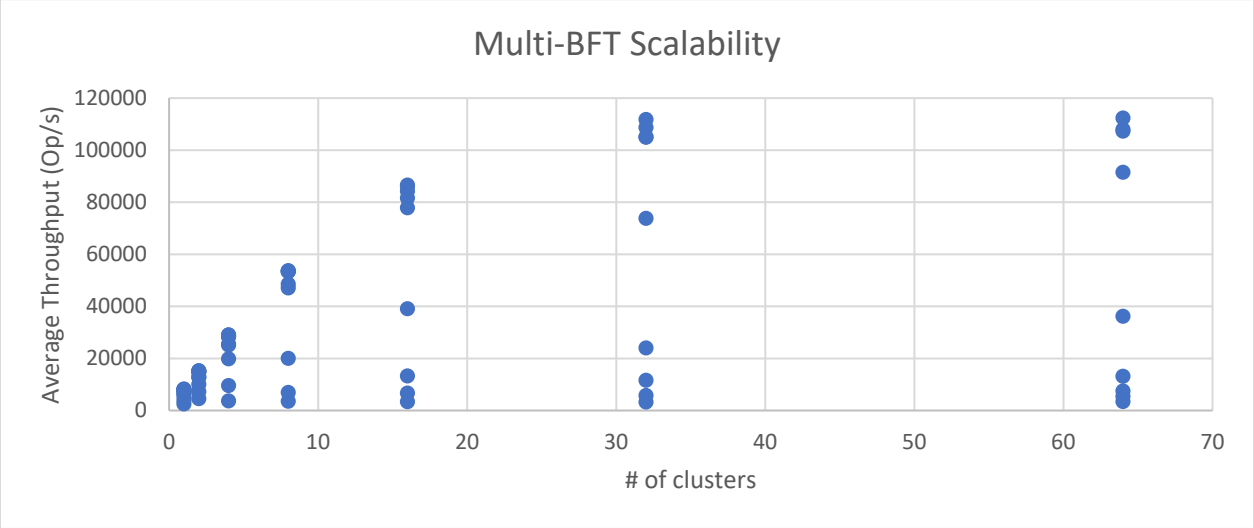
Implementation

We implemented the system in Go using Smart-BFT as the software library for providing BFT replication[xiv]. We used ed25519[xv] for digital signatures, and the Noise library[xvi] for network endpoint authentication and encryption. We containerized and deployed the codebase to Amazon Web Services (AWS) using Kubernetes in one region. We evaluated the throughput of the database as we scaled the input load for different system configurations.

We tested system configurations with equal numbers of shard and coordinator clusters. For each experiment we doubled the number of clusters from 1 to 64. Each cluster contained 4 nodes, enough to tolerate one byzantine fault per cluster. Coordinators were programmed to uniformly distribute keys between the fixed number of shards. Each experiment consisted of a sweep of increasing offered load from a fleet of benchmarking clients, also hosted in our Kubernetes cluster.

The benchmarking client maintained a pool of virtual users with one client per coordinator cluster. Each client swept from 1024 and 1048576 virtual users, doubling the load for every sample in the experiment. Each sweep increment lasted for 60 seconds. Each virtual user performed a blocking GET request from a coordinator using a random 32-byte key. The clients recorded the timestamp and response latency for each request. We aggregated the results and calculated the average throughput for each sample.

Results



Results: Multi-BFT Scalability Plot

We plotted the average throughput for each sample in the Figure above. The X axis shows the number of shard and coordinator clusters used for the system configuration in the experiment. The Y axis shows the average total throughput in operations per second for all the clients in the sample. Each data point represents a sample with a different number of virtual users per client so that the entire sweep is displayed.

The plot shows peak throughput of approximately 110K requests per second. This sample had an 0.2 second average response latency. The plot shows peak throughput occurred with 32 shard and coordinator clusters. The results show diminishing returns from adding additional clusters, with a plateau after 32 clusters and 64 clusters showing a worse peak throughput.

The experimental results demonstrate the impact of cluster scaling on the average throughput of the BFT system under increasing load. As the number of clusters in each subsystem doubled from 1 to 64, the average maximum throughput showed a steady increase, with diminishing returns.

Discussion

Scalability

We first sought to determine whether we could build a scalable BFT database by leveraging multiple BFT clusters working together in parallel. We found that with our architecture, we can scale horizontally up to a certain bound. By experimentation we found that the system throughput will continue to show improvement and scale up to 32 shard and coordinator clusters. Beyond this number, the throughput plateaus and there is no further increase in performance. The quantity of clusters (up to the plateau) does not appear to significantly impact the latency of the operations. While our design does scale to a maximum total throughput and might meet the needs of a contemporary payments system, it currently has a ceiling, and the demands of the future may require greater throughput. This would require developing a more efficient inter-cluster communication protocol.

Our current protocol has overhead which scales n^2log2(n) with the number of clusters. Ideally, this overhead would not depend on n, so that overhead does not increase with the number of clusters, which would alleviate the scalability plateau from the current system. This would enable a potentially linearly scalable version of the system, comparable to Project Hamilton and PArSEC. ERC20 account-to-account balance transfers in PArSEC require eight operations so this design could support approximately 13.75 thousand balance transfer transactions per second. Whether this performance could meet the needs of a contemporary or future payments system depends on the specific requirements of the use case[xvii].

Comparative Properties

We considered the following metrics when comparing our system design to existing scalable CFT systems, in particular PArSEC:

- Security and Resiliency - the system's ability to prevent, detect and recover from unauthorized or unintended access, modification or destruction of the data,
- Operational Cost – how much the system costs to maintain and run day-to-day,
- Scalability - the system's ability to accommodate increasing transaction throughput while maintaining acceptable latency (how long each transaction must wait to complete), and
- Complexity – the up-front costs associated with building and maintaining the system, for example the need to build bespoke software or retain subject matter experts.

While Project Hamilton and PArSEC are resilient against crash faults, both are unable to withstand a malicious actor or system bug within the clusters. Our system is resilient to this class of faults and therefore could be useful for certain high-value applications where the threat model includes bugs and malicious attacks. However, the operational cost of running BFT clusters is greater than CFT. BFT clusters need 3f + 1 nodes to tolerate f faulty nodes, whereas CFT clusters only need 2f + 1. Furthermore, the overhead of BFT consensus versus CFT means that each individual cluster can process fewer requests per second. Both factors means that more nodes are required in total to achieve comparable throughput to a CFT-based system.

We demonstrated horizontal scalability up to 32 clusters with low transaction latency (0.2s) and high throughput (110k operations/second). This is less than PArSEC which achieved closer to 1.4 million op/s (assuming each transaction included 8 operations). Furthermore, PArSEC continues to scale linearly with the number of clusters at the peak demonstrated throughput, whereas our architecture plateaued at 32 clusters. While Heraclius' throughput may be sufficient for some payment systems applications, if scalability is critical, a CFT-based solution may be more desirable in lieu of the additional security guarantees of BFT. Furthermore, compared to a CFT architecture, our scalable BFT architecture is much more complex. Both implementation and maintenance are intricate, and operationalizing a system based on this prototype would be challenging.

Having considered all these metrics, for certain classes of mission-critical applications that require greater levels of risk mitigation against downtime and correctness violations, the additional costs of Heraclius might be worth the reduction in exposure to byzantine faults. However, less security critical applications that need greater scalability and lower costs may want to consider CFT-based architectures.

Future Work

Expanding the scope of the performance tests to include a wider range of data distributions, workload mixes and concurrency levels would provide a more comprehensive understanding of system behavior under diverse conditions. A deeper analysis of performance metrics with different node distributions and faulty nodes could provide valuable insight into the performance of a real-world deployment of our system.

To support more complex applications, the system would need to provide support for transactions that span shard clusters. Specifically, implementing the two-phase commit (2PC) protocol could enable compound operations by ensuring atomicity across shards. Exploring different concurrency control algorithms to support concurrent conflicting transactions would help to better understand the tradeoffs in performance for a payment system workload. By extension, implementing a smart contract programming environment at the coordinator layer could allow Heraclius to be used for a wide range of applications and bring the system to feature-parity with PArSEC. This could provide the widest range of options between a BFT or CFT architecture.

Recent advances in Zero Knowledge Proofs (ZKPs) show that it is possible to construct ZKPs with constant size instead of $\log_2(n)$ sized Merkle proofs.[xviii] These advances might provide a path to increased scalability.

Conclusion

We sought to understand whether we could build a scalable BFT key-value store by leveraging multiple BFT clusters working together. Our research demonstrates that a scalable key-value store that leverages multiple BFT clusters working together can be built to provide 110 thousand ops/s, which may be sufficient for payments applications, but scalability may be limited beyond that.

We also sought to compare the scalability profiles of our system against those of a CFT system and explore the tradeoffs between the two designs. Although CFT systems can support processing a greater number of transactions, they offer no direct protection against some types of risk (e.g., compromised nodes). In contrast, our BFT system might be capable of scaling to a level sufficient to meet the needs of a modern payments system and deliver protection against Byzantine Faults, thereby delivering greater system resiliency.

Reviewing the profiles of these two systems, each has costs and benefits. Our BFT system has high resiliency and has moderately high throughout. Such a BFT system might be desirable where there is a high value application, resiliency is paramount, and bandwidth needs are met. On the other hand, CFT systems might be desirable where scalability is paramount. Security for CFT can augmented through other means (e.g., walled gardens). Which system is preferable would depend on the use case, and neither is universally preferable for all situations.

References

A History of AWS Cloud and Data Center Outages, https://www.datacenterknowledge.com/outages/a-history-of-aws-cloud-and-data-center-outages (Apr 2024)

Artem Barger, Yacov Manevich, Hager Meir, and Yoav Tock*, A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric*, https://arxiv.org/abs/2107.06922 (Jul 2021).

Bft-SMaRt High-performance Byzantine Fault-Tolerant State Machine Replication, https://bft-smart.github.io/library/.

Daniel Bennarroch et al., *Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular*, https://eprint.iacr.org/2019/1255.pdf (2019)

Diego Ongaro and John Ousterhout, *In Search of An Understandable Consensus Algorithm (Extended Version), https://raft.github.io/raft.pdf (May 2014).*

Edwards-Curve Digital Signature Algorithm (EdDSA), https://datatracker.ietf.org/doc/html/rfc8032 (Jan 2017).

Haryadi Gunawi et al., *Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems*. Proceedings of the 16th USENIX Conference on File and Storage Technologies, https://ucare.cs.uchicago.edu/pdf/fast18-failSlowHw.pdf (Feb 2016).

James Lovejoy, Anders Brownworth, Madars Virza, and Neha Narula, *PARSEC: Executing Smart Contracts in Parallel*, https://static1.squarespace.com/static/59aae5e9a803bb10bedeb03e/t/64c90f6427f7101a41668817/1719943974451/p.pdf (Oct 2023).

James Lovejoy et al., *Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies.* Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, https://www.usenix.org/system/files/nsdi23-lovejoy.pdf (Apr 2023).

Kyle Croman, *On Scaling Decentralized Blockchains*, https://people.eecs.berkeley.edu/~dawnsong/papers/On%20Scaling%20Decentralized%20Blockchains_feb%202016.pdf (Feb 2016).

Leslie Lamport, Robert Shostak, and Marshall Pease, *The Byzantine General's Problem.* ACM Transactions on Programming Languages and Systems v4 Is. 2, p. 382 https://dl.acm.org/doi/10.1145/357172.357176 (Jul 1982).

Maofin Yin et al., *HotStuff: BFT Consensus in the Lens of Blockchain*, https://arxiv.org/abs/1803.05069 (Jul 2019).

Miguel Castro and Barbara Liskov, *Practical Byzantine Fault Tolerance*, Proceedings of the Third Symposium on Operating System Design and Implementation https://pmg.csail.mit.edu/papers/osdi99.pdf (Feb 1999).

Remediation and Guidance Hub: Falcon Content Update for Windows Hosts, https://www.crowdstrike.com/falcon-content-update-remediation-and-guidance-hub/ (July 2024).

Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, https://bitcoin.org/bitcoin.pdf (Oct 2008).

Trevor Perrin, *The Noise Protocol Framework Revision 34*, https://noiseprotocol.org/noise.pdf (July 2018).

Weizhao Tang et al., *CFT-Forensics: High-Performance Byzantine Accountability for Crash Fault Tolerant Protocols*, https://arxiv.org/abs/2305.09123 (May 2023).

---

[i] Silent Data Corruption - some errors go unnoticed, without being detected by the disk firmware or the host operating system; these errors are known as silent data corruption.
[ii] See generally *Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems* and *A History of AWS Cloud and Data Center Outages.*
[iii] See *Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies*.
[iv] See *PARSEC: Executing Smart Contracts in Parallel.*
[v] See *In Search of an Understandable Consensus Algorithm (Extended Version).*
[vi] See *CFT-Forensics: High Performance Byzantine Accountability for Crash Fault Tolerant Protocols.*
[vii] See *HotStuff: BFT Consensus in the Lens of Blockchain.*
[viii] See Bft-SMaRT: High-performance Byzantine Fault-Tolerant State Machine Replication.
[ix] See *Practical Byzantine Fault Tolerance.*
[x] See *On Scaling Decentralized Blockchains.*
[xi] Heraclius was a Byzantine general and emperor who was known for reorganizing and strengthening the imperial administration and armies, making the empire defensively stronger.
[xii] See *The Byzantine General's Problem.*
[xiii] See *Remediation and Guidance Hub: Falcon Content Update for Windows Hosts.*
[xiv] See *A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric.*
[xv] See *Edwards-Curve Digital Signature Algorithm (EdDSA).*
[xvi] See *The Noise Protocol Framework.*
[xvii] See *Hamilton: A High-Performance Transaction Processor for Central Bank Digital Currencies.*
[xviii] See *Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular*.