Chop Chop: Byzantine Atomic Broadcast to the Network Limit

Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron, *EPFL*

https://www.usenix.org/conference/osdi24/presentation/camaioni

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Chop Chop: Byzantine Atomic Broadcast to the Network Limit

Martina Camaioni Rachid Guerraoui Matteo Monti
Pierre-Louis Roman Manuel Vidigueira Gauthier Voron
Ecole Polytechnique Fédérale de Lausanne (EPFL)

Abstract

At the heart of state machine replication, the celebrated technique enabling decentralized and secure universal computation, lies Atomic Broadcast, a fundamental communication primitive that orders, authenticates, and deduplicates messages. This paper presents Chop Chop, a Byzantine Atomic Broadcast system that uses a novel authenticated memory pool to amortize the cost of ordering, authenticating and deduplicating messages, achieving "line rate" (i.e., closely matching the complexity of a protocol that does not ensure any ordering, authentication or Byzantine resilience) even when processing messages as small as 8 bytes. Chop Chop attains this performance by means of a new form of batching we call *distillation*. A distilled batch is a set of messages that are fast to authenticate, deduplicate, and order. Batches are distilled using a novel interactive protocol involving *brokers*, an untrusted layer of facilitating processes between clients and servers. In a geo-distributed deployment of 64 medium-sized servers, Chop Chop processes 43,600,000 messages per second with an average latency of 3.6 seconds. Under the same conditions, state-of-the-art alternatives offer two orders of magnitude less throughput for the same latency. We showcase three simple Chop Chop applications: a Payment system, an Auction house and a "Pixel war" game, respectively achieving 32, 2.3 and 35 million operations per second.

1 Introduction

Is an Internet computer feasible? A computer that is highly-available, decentralized, secure, universal and shared by all? Theory says yes: state machine replication (SMR) [28, 66] enables decentralized universal computation in the face of arbitrary failures [50, 68]. In practice, however, SMR's inefficiency still makes for limited throughput. At the heart of SMR lies Atomic Broadcast [24], a powerful consensus-equivalent primitive that comes with fundamental bounds [29] and constraints [32], hindering its real-world performance despite decades of extensive research [5, 9, 14, 18, 20, 47, 53, 62, 79, 80, 82] and attention from industry, where SMR powers a myriad of blockchains and ledgers [4, 35, 46, 48, 52, 58, 72–74, 77, 78].

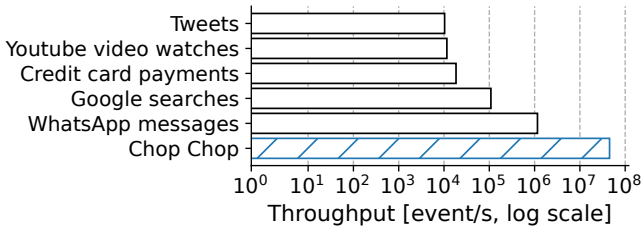


Figure 1: **Throughput of Internet-scale services.**

When deployed globally, seminal Atomic Broadcast implementations, such as BFT-SMaRt [9] and HotStuff [80], can deliver a few thousand messages per second, three orders of magnitude short of the millions of requests per second collectively handled by the Internet's largest, centralized services (Fig. 1). Pushing Atomic Broadcast's throughput into the tens of millions of messages per second seems a necessary stepping stone towards achieving an Internet-scale computer.

Towards line rate. While slow and expensive, ordering messages in Atomic Broadcast is amenable to *batching* [18]: order once, deliver in bulk. This observation motivated the development of *memory pool* (mempool) protocols [26, 34, 69], as initiated by Narwhal [26], designed to amortize ordering. This strategy proved effective, e.g., Bullshark [69] delivers in the order of 380,000 messages per second when accelerated by Narwhal. Despite this improvement, however, state-of-the-art batching still falls short of achieving *line rate*, i.e., matching the communication complexity of a protocol that does not ensure any ordering, authentication, or Byzantine resilience. In such a simplified setting, a server could simply deliver a sequence of application messages as it receives them from the network: b bits received, b bits delivered. Modern connections have enough bandwidth to receive tens of millions of application messages per second:[1] 2.5 orders of magnitude of gap still exist between Atomic Broadcast and unordered, unauthenticated dissemination. It is natural to ask

[1] Payloads as small as 12 bytes can have real-world applications (see §2.1). A 5 Gbit/s link can receive 52 millions such payloads per second.

if such a large gap is inherent to atomicity's unavoidable cost of ordering, authenticating and deduplicating messages. This paper answers in the negative, accelerating Atomic Broadcast by a further two orders of magnitude with a system that performs close to optimal efficiency, i.e., within 8% of line-rate, even when handling 40 million requests per second.

Chop Chop. We present Chop Chop, a Byzantine Atomic Broadcast system using a novel authenticated mempool. Mempools amortize the cost of ordering by having an underlying instance of Atomic Broadcast order batches. Classic methods of batching, however, fail to also amortize authentication and deduplication: each payload in a batch still carries an individual public key, signature and sequence number.

Chop Chop addresses this shortcoming with a new form of batches: *distilled batches*. Unlike a classic batch, a distilled batch contains condensed information that allows to authenticate and deduplicate its messages in bulk, much faster than in existing schemes. Distilled batches leverage the strong ordering of Atomic Broadcast to minimize redundant information.

Trustless brokers. Chop Chop produces distilled batches using a novel interactive protocol involving *brokers*, a layer of facilitating processes between clients and servers. Distilled batches are faster for servers to receive and process, but expensive for brokers to produce: distillation is interactive and relies on expensive cryptographic operations for brokers.

Importantly, however, incorrectly distilled batches are visibly malformed. As such, brokers can be *untrusted*: good brokers take load off the servers; bad ones cannot compromise the system's safety. Servers are exposed to every message in the system, bottleneck easily, and only a threshold of them can be compromised before the system loses safety. Brokers, instead, can be spun up by anyone, outside of Chop Chop's security perimeter, to meet the load produced by clients.

Evaluation. We evaluate Chop Chop in a cross-cloud, geo-distributed environment including 320 medium-sized AWS EC2 machines and 64 OVH machines. We simulate up to 257 million clients and consider 12 experimental environments. Setting up each environment requires the installation of 13 TB of synthetic workload. A naive installation using `scp` from a single machine would take 68 hours. We designed `silk`, a one-to-many peer-to-peer file transfer tool optimized for high latency connections, to install the files in 30 minutes instead.

We compare Chop Chop's throughput and end-to-end latency against its baselines in multiple real-world scenarios including server failures, adverse network conditions, and applications running. In all scenarios, Chop Chop's throughput outperforms its closest competitor by up to two orders of magnitude, with no penalty in terms of latency. When put under stress, Chop Chop orders, authenticates and deduplicates upwards of 43,600,000 messages per second with a mean latency of 3.6 seconds. Except under the most adverse network conditions and proportions of faulty clients, Chop Chop still achieves millions of operations per second.

Applications. Unlike most Atomic Broadcast implementations [9, 26, 69, 80], Chop Chop does not offload authentication and deduplication to the application. This allows Chop Chop-based applications to focus entirely on their core logic without ever engaging in expensive, and easy to get wrong, cryptography. To showcase this, we implement three simple applications to evaluate on top of Chop Chop: a Payment system, an Auction house and an instance of the game "Pixel war". These three simple applications (300 lines of logic) work effectively with messages as small as 8 bytes, further underlying the communication overhead represented by public keys, signatures and sequence numbers in non-distilled systems. Both Payments and Pixel war inherit Chop Chop's throughput, respectively processing over 32 and 35 million operations per second. Even the Auction house, which is single-threaded, achieves 2.3 million operations per second. (These applications are meant as examples, and further optimization is beyond the scope of this paper.)

Contributions. We identify authentication and deduplication as the main bottlenecks of batched Atomic Broadcast; we introduce distilled batches to extend the amortizing properties of batching to authentication and deduplication; we present distillation, an interactive protocol to produce distilled batches, and identify the opportunity to offload it to an untrusted set of brokers; we implement Chop Chop, a Byzantine Atomic Broadcast system that takes advantage of distillation through an authenticated mempool; we thoroughly evaluate Chop Chop, improving state-of-the-art Atomic Broadcast throughput by two orders of magnitude, maintaining near line-rate performance up to 40 million requests per second; we showcase Chop Chop through a Payment system, an Auction house and an instance of the "Pixel war" game, respectively achieving 32, 2.3 and 35 million operations per second.

Roadmap. §2 introduces Atomic Broadcast, discusses classic batching mechanisms and highlights the cost of authenticating and deduplicating messages in the resulting batches. §3 presents distilled batches and introduces a simplified failure-free version of Chop Chop's protocol. §4 describes Chop Chop's fault-tolerant protocol in detail. §5 discusses Chop Chop's implementation. §6 discusses Chop Chop's empirical evaluation, highlighting the challenges of such large scale experiments. We summarize related work in §7 and future work in §8. Appendix A describes Chop Chop's artifact. The full correctness proof of Chop Chop is available online [15].

2 Atomic Broadcast

In an Atomic Broadcast system [19], *clients* broadcast messages that are delivered by *servers*.

Properties [13]. Correct servers deliver the same messages in the same order (*agreement*). Messages from correct clients are eventually delivered (*validity*). Spurious messages cannot be attributed to correct clients (*integrity*). No message is delivered more than once (*no duplication*).

2.1 Cost of Atomic Broadcast

Informally, Atomic Broadcast's most distinctive property, agreement, is also the most challenging to satisfy. Correct servers must coordinate to *order* messages without compromising liveness. A great deal of research effort has been put in developing ordering techniques, optimizing for latency [47, 56] or communication complexity [55, 63].

Integrity and no duplication, instead, allow for simple solutions. Clients can ensure integrity by *authenticating* their messages using digital signatures: servers simply ignore incorrectly authenticated messages. For no duplication, clients can tag each message with a strictly increasing *sequence number*: after ordering, servers discard old messages as replays.

Both techniques—we call them *classic authentication* and *classic sequencing*—are non-interactive, easy to implement, and agnostic of the protocol employed to order messages. Arguably due to the simplicity and effectiveness of classic authentication and sequencing, most Atomic Broadcast implementations overlook integrity and no duplication entirely: they offload authentication and sequencing to the application, focusing on the more challenging task of ordering.

Batching for ordering. Lacking an efficient technique to minimize its complexity, ordering could be Atomic Broadcast's main bottleneck.[2] The well-known strategy of *batching*, however, is both general and effective at amortizing the agreement cost of an Atomic Broadcast implementation [18, 68].

Broadly speaking, batching is orchestrated by a *broker* as follows [26].[3] Over a small window of time, the broker collects multiple client-issued messages in a batch, which it disseminates to the servers; the broker then submits to an underlying instance of Atomic Broadcast a cryptographic hash of the batch it collected; upon delivering the hash of a batch from Atomic Broadcast, a server retrieves the batch, and delivers to the application all the messages it contains. Because the size of a hash is constant, the cost of ordering a batch does not depend on its size: as batches become larger, the cost of ordering each message goes to zero. In practice, batching can effectively eliminate the cost of ordering in any real-world implementation of Atomic Broadcast.

Cost of integrity and no duplication. Batching does not efficiently uphold integrity and no duplication. Regardless of how many messages are batched together, the cost of classic authentication and sequencing stays constant: one public key, one signature and one sequence number for each message.

In practice, these costs dominate the computation and communication budget of a batched Atomic Broadcast system (see §3.2). On the one hand, signatures are among the most CPU-intensive items in the standard cryptographic toolbox, dwarfing in particular symmetric primitives such as hashes and ciphers. On the other, public keys, signatures and sequence numbers can easily account for the majority of a batch's size.

To illustrate these costs, consider the example of a payment system. A payment operation requires three fields: sender, recipient, and amount. Sender and recipient fit in 4 B each if the system serves less than 4 billion users. Amount needs 4 B for payments between 1 cent and 40 millions. Hence, a payment can be encoded in just 12 B. Using public keys to identify sender and recipient (2×32 B using Ed25519 [8, 40]) and attaching a signature (64 B) and a sequence number (8 B) to each message inflates payloads to 140 B. For payments, *91% of the bandwidth is spent on integrity and no duplication*.

2.2 Existing Mitigations

Chop Chop integrates the two following techniques to reduce the bandwidth and CPU cost of authentication.

Short identifiers. Repeated public keys consume a significant slice of a server's communication budget. A workaround is to have servers store public keys in an indexed *directory* [2]. Upon first joining the system, a client announces its public key via Atomic Broadcast to *sign up*. Upon delivering a sign-up message, a server appends the new public key to its directory. The same public key appears at the same position in the directory of all correct servers thanks to Atomic Broadcast's agreement. Having signed up, a client uses its position in the directory as identifier instead of its public key.

In the previous example of a payment system, using such identifiers reduces a payment size by 40%, from 140 B to 84 B. However, a signature per payment must still be transmitted.

Pooled signature verification. Authenticating a batch by verifying its signatures is a computationally intensive task for a server [18, 71]. However, Red Belly [23] and Mir [71] showed that not all servers need to authenticate all batches. Indeed, assuming at most f faulty servers, a broker optimistically asks only $f+1$ servers to authenticate a batch to be certain to reach at least one correct server. If $f+1$ servers do not reply by a timeout, the broker extends its request to f additional servers, thus reaching at least $f+1$ correct servers.

A correct server that authenticates a batch sends back to the broker a *witness shard*, i.e., a signed statement that the batch is correctly signed. The broker aggregates $f+1$ identical shards into a *witness*, which it sends to the other $2f$ servers. Because every witness contains at least one correct shard, the servers can trust the witness instead of verifying the batch.

Assuming $3f+1$ servers, this technique shaves up to two-thirds off the system's authentication complexity.

3 Distilled Batches

Chop Chop's main contribution is *distillation*, a set of techniques aimed at extending the amortizing properties of batches to authentication and sequencing.

[2] Byzantine Atomic Broadcast among n participants cannot be achieved with a bit complexity smaller than $\Theta(n^2)$ [29].

[3] In the literature, servers usually play the role of brokers. As we discuss in §4, however, Chop Chop minimizes its load on the servers by offloading brokerage to a separate, trustless set of processes.

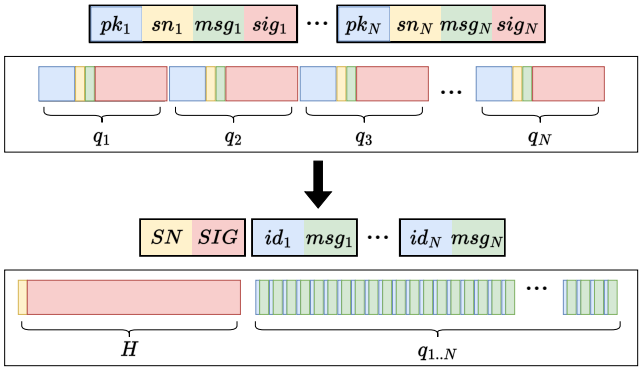


Figure 2: **Full distillation in action.** With classic authentication and sequencing, each payload q_i contains a public key pk_i, a sequence number sn_i, a message msg_i and a signature sig_i. In the fully distilled case, each q_i reduces to just id_i and msg_i: one header H, composed of one aggregate sequence number SN and one aggregate signature SIG, is sufficient for the entire batch. Bars are to scale if small messages are broadcast using Ed25519 for signatures and BLS12-381 for uncompressed multi-signatures: sn_i and SN are 8 B, msg_i is 8 B, pk_i is 32 B, sig_i is 64 B, SIG is 192 B.

Background: multi-signatures. Chop Chop makes use of multi-signature schemes [39] to authenticate batches. Secret keys produce signatures that can be verified against the corresponding public keys. Public keys and signatures, however, can be *aggregated*. Let $(p_1,r_1),\dots,(p_n,r_n)$ be distinct key pairs, and $s_1,\dots,s_n$ be signatures produced by $r_1,\dots,r_n$ on the *same* message m: $p_1,\dots,p_n$ (resp., $s_1,\dots,s_n$) can be aggregated into a constant-sized aggregate public key p (resp., aggregate signature s).

Remarkably, s can be verified in constant time against p and m [10, 57]. Chop Chop uses BLS multi-signatures [10] which can be aggregated cheaply and non-interactively: even a non-signing process can compute p (resp., s) once provided with $p_1,\dots,p_n$ (resp., $s_1,\dots,s_n$) by computing a single multiplication over an elliptic curve.

3.1 Distillation at a Glance

In brief, distillation aims to produce *distilled batches*. A distilled batch has some of its signatures (resp., sequence numbers) replaced by an *aggregate signature* (resp., *aggregate sequence number*). When maximally successful, distillation produces a *fully distilled batch*, where all signatures (resp., sequence numbers) have been replaced by a *single* aggregate signature (resp., sequence number). As we discuss below, distilled batches are vastly cheaper for servers to receive and process. Fig. 2 depicts the effect of distillation on a batch.

Full distillation (failure-free). For pedagogical purposes, we introduce distillation under the assumption that all processes are correct. We detail Chop Chop's fault-tolerant distillation techniques in §4.2, optimized and adapted to the Byzantine setting. As in the classic batching case, a set $\chi_1,\dots,\chi_b$ of clients submit their messages $m_1,\dots,m_b$ to a broker β. Each χ_i selects for its message m_i a sequence number k_i (greater than any sequence number it previously used), then sends (k_i,m_i) to β. Upon receiving all (k_i,m_i)-s, β computes the aggregate sequence number

$$k = \max_i k_i$$

then builds the *batch proposal*

$$B = [(x_1,k,m_1),\dots,(x_b,k,m_b)]$$

where x_i is χ_i's numerical identifier in the system (see §2.2). β then sends B back to every χ_i. Upon receiving B, χ_i produces a multi-signature s_i for the hash $H(B)$ of B, which it sends back to β. Having collected all multi-signatures, β computes the aggregate signature

$$s = \prod_i s_i$$

In doing so, β obtains the fully distilled batch

$$\tilde{B} = [s,k,((x_1,m_1),\dots,(x_b,m_b))]$$

Upon receiving $\tilde{B}$, any server now can: compute B by inserting k between each (x_i,m_i); compute $H(B)$; use each x_i to retrieve χ_i's public key p_i from its directory; compute the aggregate public key

$$p = \prod_i p_i$$

and finally verify s against p and $H(B)$.

Distillation outcome. Having engaged with β to distill the batch, every χ_i multi-signs the *same* message $H(B)$ and updates its sequence number to the *same* k. This allows β to authenticate and sequence all of $\tilde{B}$ using s and k only.

Distillation safety. The proposed distillation protocol has no safety drawback. First, because (x_i,k,m_i) appears in B, χ_i still gets to authenticate m_i. Intuitively, χ_i's multi-signature on $H(B)$ publicly authenticates *whatever message in B is attributed to χ_i*, m_i in this case. Second, because $k \geq k_i$, k is still a valid sequence number for m_i. Sequence number distillation might cause χ_i to skip some sequence numbers whenever any χ_j issues some $k_j > k_i$. Contiguity of sequence numbers, however, is not a requirement for deduplication. As with classic sequencing, χ_i produces—and servers deliver—messages with strictly increasing sequence numbers; servers disregard all other messages as replays.

3.2 Distillation Microbenchmark

Having discussed how distilled batches are produced, we now estimate the significance of their effect by means of a back-of-the-envelope calculation and a simple microbenchmark on AWS. Consider a setting where 100 million clients broadcast 8-byte messages, e.g., to issue payments (see § 2.1). We compare *classic authentication and sequencing*, where clients are identified by their public keys, messages are individually signed and sequenced, against *fully distilled batches* where

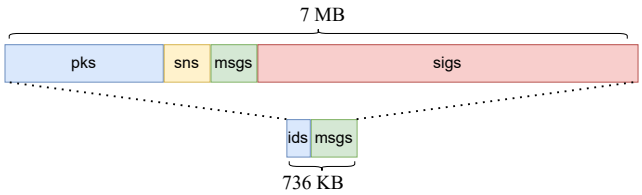


Figure 3: **Full distillation of a batch of 65,536 payloads (sizes to scale).** The aggregate signature and aggregate sequence number do not appear as a result of their small size.

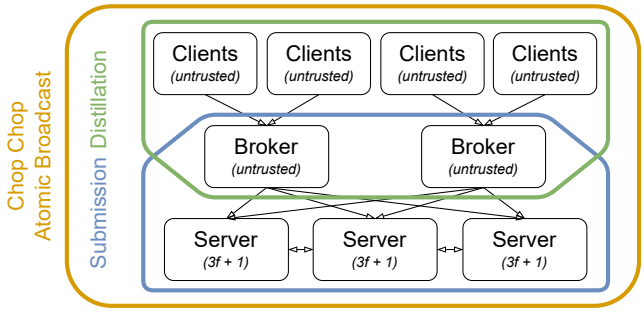


Figure 4: **Chop Chop architecture.**

clients are identified by a numerical identifier and each batch contains only one aggregated signature and sequence number. We use Ed25519 [40] for signatures (32 B public keys, 64 B signatures) and BLS12-381 [12] for multi-signatures (192 B uncompressed signatures). We use uncompressed BLS multi-signatures to save computation time at the cost of storage space (96 B compressed vs. 192 B uncompressed).

Communication complexity. Payloads are 112 B per message in the classic case (32 B of public key, 8 B of sequence number, 8 B of message, 64 B of signature) vs. 11.5 B in the fully distilled case (28 bits = 3.5 B of identifier to represent 257M clients, 8 B of message). Assuming batches of 65,536 messages (Fig. 3), classic batches are exactly 7 MB long, while fully distilled batches are 736 KB long including aggregate signature and sequence number.

Computation complexity. Running at maximum load, an Amazon EC2 c6i.8xlarge instance authenticates 16.2 ± 0.4 classic batches per second using Ed25519's batch verification for 65,536 signatures. The same machine authenticates 457.1 ± 0.3 fully distilled batches per second: each authentication requires the aggregation of 65,536 BLS12-381 public keys and the verification of one BLS12-381 multi-signature.

Summary. By the order-of-magnitude calculations above, fully distilled batches hold the promise to reduce the costs of authentication and sequencing by a factor 9.7 for network bandwidth, and 28.2 for CPU. Chop Chop aims to deliver on that promise for a real-world fault-tolerant system.

4 Chop Chop

This section overviews Chop Chop's architecture, Chop Chop's protocol, and provides arguments for its correctness.

Overview. Chop Chop involves three types of processes (Fig. 4): broadcasting clients, delivering servers and a layer of broadcast-facilitating brokers between them. Servers run an Atomic Broadcast instance among themselves, to which brokers submit messages. Chop Chop is agnostic to the implementation of Atomic Broadcast used by the servers. On top of the provided broker-to-server Atomic Broadcast, Chop Chop implements a much faster client-to-server Atomic Broadcast: clients submit messages to the servers, aided by brokers.

Chop Chop's protocol unfolds in two phases: *distillation* (§4.2) and *submission* (§4.3). In the distillation phase, clients interact with a broker to gather their messages in a distilled batch (see §3). In the submission phase, the broker disseminates the distilled batch to the servers and submits the batch's hash to the server-run instance of Atomic Broadcast. Upon delivering its hash from Atomic Broadcast, servers retrieve the batch and deliver its messages. Chop Chop's contributions mainly focus on the distillation phase. Chop Chop's submission strategy closely resembles prior batch-based Atomic Broadcast implementations [26, 34, 69].

4.1 Architecture and Model

Chop Chop augments the architecture of a classic Atomic Broadcast, as described in §2, with novel brokers.

Clients and servers. *Clients* broadcast messages to a (distinct) set of *servers*. We assume that less than one third of servers can be faulty and behave in an arbitrary manner, i.e., be Byzantine [50], while all clients can be faulty. For simplicity, servers form a fixed set that is known by all correct processes at system startup. Chop Chop can be extended for reconfiguration thanks to its modular use of Atomic Broadcast [9, 49] (Fig. 4). Clients issue messages after broadcasting their public keys to the system (see §2.2).

Brokers. We discussed in §3 how both classic and distilled batches are assembled by a broker. The role of brokers is traditionally taken by servers. Given the additional strain put on brokers by Chop Chop's interactive distillation protocol, however, having servers be brokers would result in a waste of scarce, trusted resources. Importantly, however, distillation is *trustless*. On the one hand, agreement rests entirely on Chop Chop's underlying Atomic Broadcast instance, for which brokers are only clients. On the other hand, as we argue in §§4.2 and 4.4.1, a faulty broker cannot compromise integrity or no duplication: distilled batches are publicly authenticated, and correct clients cannot be tricked into using stale sequence numbers. Hence, *brokers need no trust*: a broker either does its job correctly or produces distilled batches that are visibly malformed, and easily discarded by all correct servers.

This observation is of paramount importance to the performance of Chop Chop: *because distillation is heavy but trustless, brokers should be distinct from servers*. Along with

clients and servers, we thus assume a third, *independent set of brokers*, sitting between clients and servers, to accelerate Atomic Broadcast by assembling client messages in distilled batches. We assume that at least one broker is correct; the system loses liveness but not safety if all brokers are faulty.

Network. Chop Chop guarantees that the batches collected and submitted to servers by correct brokers are well-formed even in asynchrony, but achieves full distillation when the network is synchronous (see §4.2). Chop Chop inherits the network requirements of its underlying Atomic Broadcast.

4.2 Distillation Phase

We introduced in §3 a simplified, failure-free distillation protocol. This section describes how Chop Chop renders distillation tolerant to arbitrary failures and improves its performance via a sequence of improvements, each addressing a shortcoming of the simplified protocol. The complete fault-tolerant protocol of Chop Chop is depicted in Fig. 5.

In the failure-free distillation protocol: clients $\chi_1, \ldots, \chi_b$ send their messages $m_1, \ldots, m_b$, with sequence numbers $k_1, \ldots, k_b$ (#2) to a broker β (Fig. 5, #1); β identifies the maximum submitted sequence number k and builds a batch proposal $B = [(x_1, k, m_1), \ldots, (x_b, k, m_b)]$ (#3); β disseminates B to $\chi_1, \ldots, \chi_b$ (#4); each χ_i produces a multi-signature s_i on $H(B)$ (#5), which it sends to back β (#6); β aggregates $s_1, \ldots, s_n$ into an aggregate s, thus producing a fully distilled batch $\tilde{B} = [s, k, ((x_1, m_1), \ldots, (x_b, m_b))]$ (#7).

Background: Merkle trees. Chop Chop uses Merkle trees [59] to hash batches. An l-element vector $z_1, \ldots, z_l$ is hashed into a *root* r, used as commitment. For each i, z_i's value can be proved by means of a proof of inclusion p_i, verifiable against r and z_i. Proofs of inclusions are $O(\log l)$ in size and are verified in $O(\log l)$ time.

What if a broker forges messages? A faulty β could try to falsely attribute to some χ_i a message $m_i' \neq m_i$. β could do so by replacing m_i with m_i' in B, then having χ_i sign $H(B)$, thus implicitly authenticating m_i'. This is easily fixed by having χ_i check that m_i correctly appears in B before signing $H(B)$.

Can a broker avoid sending the entire batch? A clear inefficiency of the simplified protocol is that β has to convey all of B back to each χ_i. This is fixed using Merkle trees. Upon assembling B, β computes the Merkle root r of B, along with the Merkle proof p_i for each (x_i, k, m_i) in B. Instead of sending B to all clients, β just sends r, k and p_i to each χ_i. Upon receiving r, k and p_i, χ_i checks p_i against r and (x_i, k, m_i), producing s_i on r only if the check succeeds. If χ_i signs r, then (x_i, k, m_i) is necessarily an element of B. Importantly, however, β could inject $(x_i, k, m_i' \neq m_i)$ somewhere else in B, while still providing χ_i only with the proof for (x_i, k, m_i). This is solved by having servers ignore every distilled batch where two or more messages are attributed to the same client. This way, if χ_i signs r, then either m_i is the only message in B attributed to χ_i, or $\tilde{B}$ is rejected by all servers as malformed: either way, integrity is upheld.

What if a client does not multi-sign? Under the assumption that $\chi_1, \ldots, \chi_b$ are correct, β can safely wait until it collects all $s_1, \ldots, s_b$. This policy is clearly flawed in the Byzantine setting: a single crashed client can prevent β from ever aggregating s. Furthermore, lacking an assumption of synchrony, β cannot exclude from $\tilde{B}$ those clients that do not sign r by some timeout: consistently slow clients would always be excluded, and validity would be lost. This issue is fixed by the fallback mechanism introduced in the following.

Fault-tolerant distillation. Upon first sending (k_i, m_i) to β (#2), χ_i also sends an individual, non-aggregable signature t_i for (x_i, k_i, m_i), which β stores. β then waits for s_i-s on r until either all s_i-s are collected, or a timeout expires. For every s_i that ends up missing, due to χ_i being crashed or delayed, β attaches (k_i, t_i) to $\tilde{B}$. Upon receiving $\tilde{B}$, a server first checks each individual signature t_i against the corresponding (x_i, k_i, m_i). The server then checks s against the public keys of the clients for which an individual signature t_i was not given, i.e., the public keys of all clients that signed r in time.

In summary: fast, correct clients who successfully produce their s_i-s in time authenticate their message by multi-signing r; slow or crashed clients still get their messages through, individually authenticated by the t_i-s that they originally produced. Full distillation is achieved whenever the network is synchronous and all clients are correct, which we argue is the case in practice for the majority of a system's lifetime. When the network is asynchronous, however, a fraction of clients might fail to produce their s_i in time, resulting in a *partially distilled batch*. At the limit where all clients fail to sign r in time, $\tilde{B}$ reduces to a classic batch, degrading server-side performance to pre-distillation levels. We underline that safety and liveness are preserved regardless of synchrony.

What if a broker replays messages? A problem introduced by the last fix is that χ_i authenticates both k_i and k as sequence numbers for m_i, allowing a faulty β to play m_i twice, hence breaking Atomic Broadcast's no duplication. This is fixed by having each client engage in the broadcast of only one message at a time. This way, while β can indeed replay m_i, it can only do so consecutively: all sequence numbers χ_i authenticates for m_i belong to a range that does not contain sequence numbers for any other message $m_{i' \neq i}$ issued by χ_i.

This observation is key to the following fix: along with the last sequence number $\bar{k}_\chi$ each client χ used, a correct server σ stores the last message $\bar{m}_\chi$ that χ broadcast; upon ordering a message m with sequence number k from χ, σ delivers m if and only if $k > \bar{k}_\chi$ and $m \neq \bar{m}_\chi$. In doing so, σ discards all consecutive replays of $\bar{m}_\chi$, thus preventing replays in general.

What if a client broadcasts too frequently? The last fix relies on clients broadcasting one message at a time. Depending on latency, a client broadcasting too frequently might

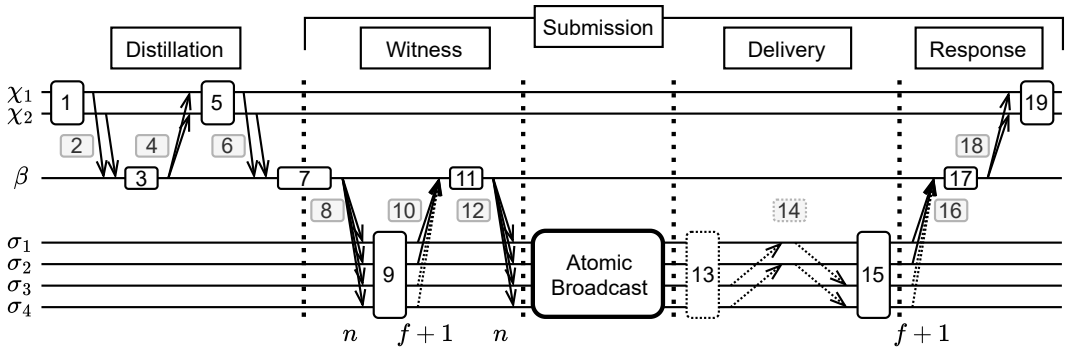


Figure 5: **Overview of the Chop Chop protocol between two clients (χ_1, χ_2), a broker (β) and four servers (σ_1–σ_4).** The protocol is comprised of 19 steps (#1–#19) and of an underlying instance of Atomic Broadcast such as BFT-SMaRt or HotStuff.

accrue an ever-growing queue of pending messages. This issue is fixed by flushing application messages in bursts, akin to Nagle's buffering algorithm for TCP.

What if a client uses the largest possible sequence number? Assuming that a finite number of bits (e.g., 64) are allocated to representing sequence numbers, a faulty client χ_m could set its k_m to the largest possible sequence number k_{max} (e.g., $2^{64}-1$). In doing so, χ_m would force all other χ_i-s to update their sequence number to k_{max}. Since correct clients only use strictly increasing sequence numbers, no χ_i could ever broadcast again: sequence numbers would run out. Proving the *legitimacy* of sequence numbers fixes this issue.

Legitimate sequence numbers. By the rule that we established, no more than one message from the same client can appear in the same batch. Moreover, correct clients always tag their messages with the smallest sequence number they have not yet used, i.e., the largest they have used plus one. By induction, we then have that unless some client misbehaves, no client ever needs to use a sequence number larger than the number of batches ever delivered by the servers: the largest sequence number any client submits to the very first batch is 0, therefore no client submits a sequence number larger than 1 to the second batch, and so on. This observation allows us to define as *legitimate* any sequence number smaller than the number of batches servers have delivered at any given time.

Legitimacy proofs. This definition of legitimacy allows for the generation of *legitimacy proofs*: upon delivering the n-th batch, a server publicly states so with a signature. By collecting $f+1$ server signatures stating that the n-th batch was delivered into a certificate l_n, any process can publicly prove that any sequence number smaller than n is legitimate.

Upon initially submitting k_i (#2), χ_i also sends to β a certificate l_n, for any $n > k_i$; β ignores client submissions that lack such certificate, except when $k_i = 0$ since no certificate is needed. Upon sending k back to all χ_i-s (#4), β attaches the highest $l_{\hat{n}}$ it collected: $l_{\hat{n}}$ proves that k is legitimate since $\hat{n} > k$. χ_i signs r (#5) only if k is proved legitimate by $l_{\hat{n}}$.

This technique ensures correct clients always use legitimate sequence numbers. Since legitimate sequence numbers grow only with the number of delivered batches, no correct client is forced to skip too far ahead, compromising its own liveness.

What if a broker crashes? If β fails to engage in the protocol, each χ_i can submit its message to any other broker.

4.3 Submission Phase

The submission phase ensures that all servers efficiently deliver a distilled batch, and that all broadcasting clients receive a proof that their messages were delivered.

Witness. Having gathered a distilled batch $\tilde{B}$ (#7), β moves on to have $f+1$ servers signs a *witness shard* for $\tilde{B}$. In signing a witness shard for $\tilde{B}$, a server σ simultaneously makes two statements. First, $\tilde{B}$ is *well-formed*: σ successfully verified $\tilde{B}$'s signatures and found all messages in $\tilde{B}$ to have a different sender. Second, $\tilde{B}$ is *retrievable*: σ stores $\tilde{B}$ and makes it available for retrieval, should any other server need it. We call a *witness* for $\tilde{B}$ the aggregation of $f+1$ witness shards for $\tilde{B}$. Because any set of $f+1$ processes includes a correct process, when presented with a witness for $\tilde{B}$ any server can trust $\tilde{B}$ to be well-formed and retrievable.

As discussed in §2.2, witnesses optimize server-side computation. Only $f+1$ servers need to engage in the expensive checks required to safely witness $\tilde{B}$. All other servers can trust $\tilde{B}$'s witness, saving trusted CPU resources.

In order to collect a witness for $\tilde{B}$, β sends $\tilde{B}$ to all servers (#8). Optimistically, β asks only $f+1$ servers to sign a witness shard for $\tilde{B}$, progressively extending its request to $2f+1$ servers upon expiration of suitable timeouts. Upon receiving $\tilde{B}$ (#9), a correct server σ stores $\tilde{B}$. If asked to witness $\tilde{B}$, σ checks that $\tilde{B}$ is well-formed and sends back to β its witness shard for $\tilde{B}$ (#10). β collects and aggregates $f+1$ shards into a witness for $\tilde{B}$ (#11), then submits $\tilde{B}$'s hash and witness to the server-run Atomic Broadcast (#12).

Delivery. Upon delivering $\tilde{B}$'s hash and witness from Atomic Broadcast (#13), a correct server σ retrieves $\tilde{B}$, either from its local storage (if it directly received $\tilde{B}$ from β at #8) or from another server (#14). Because $\tilde{B}$ is retrievable, σ

is guaranteed to eventually find a server to pull $\tilde{B}$ from. Having retrieved $\tilde{B}$ (#15), σ delivers all non-duplicate messages in $\tilde{B}$ (see §4.2 for how σ detects duplicates).

Response. Finally, σ signs a *delivery certificate*, listing the messages in $\tilde{B}$ that σ delivered. σ sends its signature back to β (#16). By agreement of Atomic Broadcast, all correct servers deliver the same subset of messages in $\tilde{B}$. As such, β is guaranteed to eventually collect $f+1$ signatures on the same delivery certificate (#17). Upon doing so, β distributes a copy of $\tilde{B}$'s delivery certificate to $\chi_1, \ldots, \chi_b$ (#18). Armed with $\tilde{B}$'s delivery certificate, a correct χ_i can publicly prove the delivery of m_i (#19) and safely broadcast its next message.

4.4 Correctness

This section summarizes Chop Chop's correctness analysis. We prove Chop Chop's correctness to the fullest extent of formal detail in an extended document available online [15].

4.4.1 Safety

The safety of Chop Chop is given by its agreement, integrity and no duplication properties (see §2).

Agreement. Chop Chop inherits agreement from its underlying, server-run instance of Atomic Broadcast. A correct server delivers messages only upon delivering the hash of a batch from the server-run Atomic Broadcast. Upon doing so, a correct server retrieves the full batch, checks its hash, and delivers all its messages in order of appearance. All correct servers deliver the same messages in the same order assuming cryptographic hashes are collision-resistant.

Integrity. A correct server only delivers messages included in a batch witnessed by $f+1$ servers, i.e., by at least one correct server. A correct server witnesses a batch only if: no more than one message in the batch is attributed to the same client; every client in the batch authenticates its message with a signature or the root of the batch's Merkle tree with a multi-signature. A correct client multi-signs the root of a batch's Merkle tree only upon receiving a proof of the inclusion of its message in the batch. As such, if a correct client multi-signs the root of a batch's Merkle tree, either the batch contains only the client's intended message or it is not witnessed. In summary, a correct server delivers a message m from a correct client χ only if χ broadcast m.

No duplication. A correct client only broadcasts one message at a time. As such, while the client might attach multiple sequence numbers to the same message (different brokers may propose different aggregate sequence numbers for the client to authenticate) the sequence numbers the client attaches to each message belong to distinct ranges. A correct server delivers client messages only in increasing order of sequence number, and ignores repeated messages. This means that a correct server delivers at most one message from each sequence number range. In summary, no server delivers a correct client's message more than once.

4.4.2 Liveness

The liveness of Chop Chop is given by its validity property.

Validity. If a correct client submits its message to a correct broker, the message is guaranteed to eventually be delivered by all correct servers: even if the client fails to engage in distillation in a timely manner, its message is still included in a batch which gets disseminated, witnessed and delivered by all correct servers. Faulty brokers can clearly refuse to service (specific) clients. Upon expiration of a suitable timeout, however, a correct client submits its message to a different broker. As we assume that at least one broker is correct, all correct clients are eventually guaranteed to find a correct broker and get their messages delivered by all correct servers.

4.4.3 Other Attacks

As we outlined in §§4.4.1 and 4.4.2, Chop Chop satisfies all properties of Atomic Broadcast. In this section, we consider other attacks an adversary might deal to impair Atomic Broadcast's performance and fairness [43] in Chop Chop.

Denial of service. A faulty broker may refuse to service clients, thus forcing them to fall back on other brokers, increasing latency. A faulty broker may also submit deliberately non-distilled batches to servers to force them to waste trusted resources to receive and verify individual signatures. While handling DoS is beyond the scope of this paper, Chop Chop is amenable to accountability mechanisms [36]. Brokers could be asked to stake resource to join the system. Correct, high-performance brokers could be rewarded, akin to gas fees in Ethereum [78]. Brokers that accrue a reputation of misbehavior or slowness could be banned and lose their initial stake.

Front-running. A faulty broker might impact fairness by front-running messages of interest [25, 83]. While front-running resistance is beyond the scope of this paper, Chop Chop is compatible as-is with existing mechanisms to mitigate or prevent front-running, most notably schemes that have clients submit encrypted messages whose content is revealed only after delivery [60, 81]. Importantly, these encrypt-order-reveal schemes could be selectively employed only for those messages that are vulnerable to front-runs, e.g., messages used for stock trading [65]. Maintaining Chop Chop's throughput while providing quorum-enforced fairness for every message [82] opens a valuable future avenue of research.

5 Implementation Details

A straightforward implementation of the protocol we presented in §4 would not achieve the throughput and latency we observe in §6. In this section, we discuss some of the techniques and optimizations required on the way to practically achieving Chop Chop's full potential. (Many optimizations are however left out due to space constraints).

Code. Chop Chop is implemented in Rust, totaling 8,900 lines of code. The main libraries Chop Chop depends on

are: `tokio` 1.12 for an asynchronous, event-based runtime; `rayon` 1.5 for worker-based parallel computation; `serde` 1.0 for serialization and deserialization; `blake3` 1.0 for cryptographic hashes; `ed25519-dalek` 1.2 for EdDSA signatures on Curve25519 [40]; `blst` 0.3.5 for multi-signatures on the BLS12-381 curve [12]. Chop Chop also depends on in-house libraries: `talk` (9,800 lines of code) for basic distributed computing and high-level networking and cryptography; `zebra` (7,100 lines of code) for Merkle-tree based data structures.

5.1 Broker

The goal of a Chop Chop broker is to produce batches as distilled as possible (to minimize server load), as large as possible (to amortize ordering), and as quickly as possible (to minimize latency). Our target is for a broker to assemble one fully distilled batch of 65,536 messages ($\sim$ 736 KB, see Fig. 3) per second, with a 1 second distillation timeout.

Reliable UDP. Short-lived TCP connections between broker and clients are easier to work with, but unfeasible for the broker to handle. Assuming an end-to-end broadcast time of up to 10 seconds, the broker would need to maintain upwards of 600,000 simultaneous TCP connections, which preliminary tests immediately proved unfeasible on the hardware we have access to. This makes UDP the only option for client-broker communication. However, UDP lacks the reliability properties of TCP, and tests showed non-negligible packet loss even within the same AWS EC2 availability zone. As we discussed in §4.2, message loss immediately translates to partial distillation. We address this issue by means of an in-house, ACK-based, message retransmission protocol based on UDP that also smoothens the rate of outgoing packets.

EdDSA batch verification. To avoid spoofing, all client messages are authenticated with signatures. At the target rate, however, individually verifying each signature is unfeasible for a broker. Luckily, `ed25519-dalek` allows for more efficient batched verification. A broker buffers the client messages it receives and authenticates them in batches.

Tree-search invalid multi-signatures. Clients contributing to the same batch produce matching multi-signatures for the batch's root. At the target rate the broker cannot independently verify each multi-signature. We tackle this problem by gathering multiple matching multi-signatures on the leaves of a binary tree: internal nodes aggregate their children. For each tree, the broker verifies the root multi-signature, recurring only on the children of an invalid parent. This allows to identify invalid multi-signatures in logarithmic time while enabling batched verification in the good case.

Caching legitimacy proofs. Clients justify their sequence numbers with legitimacy proofs. Again, the broker cannot verify each proof in time. We address this problem by having the broker verify a legitimacy proof only if higher than the highest it previously observed. As a result, a faulty client might get away with submitting an invalid legitimacy proof but, importantly, not an illegitimate sequence number.

5.2 Server

The goal of a Chop Chop server is to process distilled batches as quickly as possible without overflowing its memory.

Batch garbage collection. Servers update each other on which batches they delivered. A server garbage-collects a batch, both messages and metadata, as soon as it is delivered by all other servers. We underline that, even if a single server fails to deliver a batch, the others cannot garbage-collect it as the slow server might be correct. This is an inherent limitation of Atomic Broadcast: agreement without synchrony can be ensured only in the infinite-memory model.

Identifier-sorted batching. No two messages from the same client must appear in the same batch. To simplify processing, brokers sort the messages in a batch by client identifier. Servers reject batches whose identifiers are not strictly increasing, thus verifying that all identifiers are distinct in constant size and in linear time. Sorting messages by identifier also enables parallel deduplication: messages are split by identifier range, chunks are deduplicated independently.

6 Evaluation

We evaluate Chop Chop focusing on the following research questions (RQs): What workload can Chop Chop sustain (§6.3)? What are the benefits of Chop Chop's distillation (§6.4)? How does Chop Chop scale to different numbers of servers (§6.5)? How efficiently does Chop Chop use resources overall (§6.6)? How does Chop Chop perform under adverse conditions, such as server failures (§6.7)? What performance can applications achieve using Chop Chop (§6.8)?

6.1 Baselines

We compare Chop Chop against four baselines:

- HotStuff [80]: an Atomic Broadcast protocol designed for high-throughput (written in C++);
- BFT-SMaRt [9]: an Atomic Broadcast protocol, similar to PBFT [18], designed for low-latency (written in Java);
- Narwhal-Bullshark: the DAG-based Atomic Broadcast protocol Bullshark [69] with the state-of-the-art high-throughput mempool Narwhal [26] (written in Rust);
- Narwhal-Bullshark-sig: akin to Narwhal-Bullshark but with Narwhal modified to authenticate messages, thus matching Chop Chop's guarantees.

We deploy Chop Chop with two distinct underlying Atomic Broadcast protocols (Fig. 5): HotStuff and BFT-SMaRt.

HotStuff and BFT-SMaRt. Evaluating HotStuff and BFT-SMaRt allows us to assess the base performance of an Atomic Broadcast protocol and determine how much acceleration Chop Chop provides. We evaluate Chop Chop on top of the same implementations of HotStuff [22] and BFT-SMaRt [21] we benchmark against. These implementations

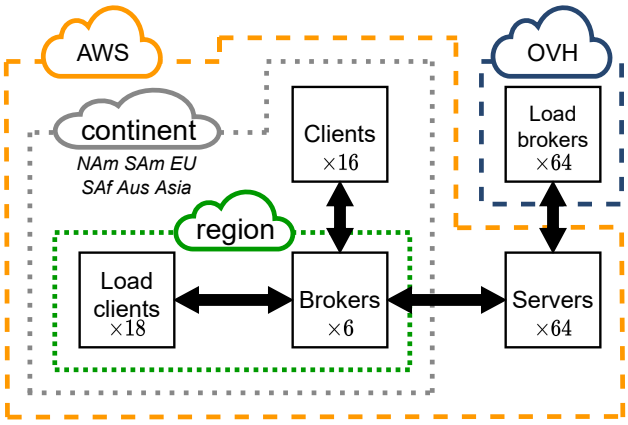


Figure 6: **Cross-cloud deployment summary.**

are production-ready and do not use state-of-the-art mempool protocols, only some basic form of batching. When evaluated stand-alone, each message in these systems includes 80 B of header composed of a client identifier (8 B), a sequence number (8 B), and a signature (64 B) verified by the servers. Both systems use batches of 400 messages, i.e., of 34.4 KB.

Narwhal-Bullshark. As a state-of-the-art mempool, Narwhal is a close point of comparison for Chop Chop. Servers in Narwhal scale out following a primary-workers model: each server is paired with one or several workers into a server group. Similarly to Chop Chop, Narwhal greatly accelerates its underlying Atomic Broadcast (here, Bullshark). Unlike Chop Chop, however, Narwhal leaves the responsibility of authenticating and deduplicating messages to the application.

Narwhal-Bullshark-sig. For a better comparison, we also benchmark Narwhal-Bullshark-sig: Narwhal-Bullshark where messages are authenticated by Narwhal in a state-of-the-art way, i.e., using batched, multi-core Ed25519 signature verification. Each message includes an 80 B header as for HotStuff and BFT-SMaRt. As for Narwhal-Bullshark, the remaining parameters are the default ones, e.g., 500 KB batches.

6.2 Setup

Unless otherwise specified—in §§6.5 and 6.6—the Chop Chop benchmarks involve 64 c6i.8xlarge AWS servers, of 32 Intel vCPUs each, geo-distributed across 14 regions. Brokers assemble, and servers process, batches of 65,536 messages. Each message is 8 B in length, resulting in 736 KB batches (Fig. 3). Baselines always use the same set of server machines as their Chop Chop counterpart. All experiments run with maximum resilience, e.g., the system survives 21 faulty servers out of 64. Fig. 6 overviews the used deployment.

Matching trusted and total resources. Unlike its baselines, Chop Chop leverages *untrusted* resources, brokers, to boost its performance. Lacking a well-defined conversion between trusted and untrusted resources, two extremes can be taken to compare Chop Chop with its baselines: we can either match trusted resources, e.g., same number of Chop Chop servers as Narwhal workers, or match total resources, e.g., same number of servers and brokers in Chop Chop as workers in Narwhal.

Intuitively, the first approach considers untrusted resources to be free while the second considers untrusted resources to be as costly as trusted resources. We use the first approach in §§6.3 to 6.5, 6.7 and 6.8 to stress Chop Chop, provisioning the system with enough brokers to bottleneck servers. We use the second approach in §6.6 to assess how efficiently Chop Chop uses its hardware resources, trusted or not.

Load clients and load brokers. We show in §6.3 that Chop Chop servers handle up to 43.6 million operations per second with an average latency of 3.6 seconds. To produce this level of workload, a real-world deployment would require over 700 brokers, each handling around 200,000 clients broadcasting back-to-back thus totaling hundreds of millions of machines. As we cannot experiment at such a scale, we introduce two new actors: *load clients* and *load brokers*. (In the rest of this section, "brokers" and "clients" denote real brokers and real clients; the term "load" is always used explicitly.)

Load clients connect to brokers and simulate thousands of concurrent client requests. Most system evaluation typically use this approach to stress the system and measure latency. However, we explicitly separate clients from load clients in this evaluation. Clients run on very small machines—less powerful than most smartphones—to provide more accurate end-to-end latency measurements. We similarly split clients from load clients in all baseline runs.

Load brokers are unique to Chop Chop. Even using load clients, we could not deploy enough brokers to bottleneck Chop Chop's servers. Load brokers work around this limitation, submitting batches of pre-generated messages directly to the servers. Free from interactions with clients and expensive cryptography, a load broker puts on the servers a load equivalent to that of tens of brokers working at full capacity.

Using load clients and load brokers, we manage to show that brokers can quickly generate large batches of messages, and servers can process large numbers of batches.

Cross-cloud deployment. All servers are deployed on AWS, balanced across 14 regions: Cape Town, São Paulo, Bahrain, Canada, Frankfurt, Northern Virginia, Northern California, Stockholm, Ohio, Milan, Oregon, Ireland, London, and Paris. For system sizes of 8 in §6.5, we distribute servers across the first 8 regions from the list, which constitute the most adversarial setup with the highest pairwise latency.

Load brokers are placed in a separate cloud provider, OVH, for two purposes. First, it provides a better representation of Internet load than a single-cloud deployment. AWS operates under its own AS so any AS peering bottlenecks would be bypassed by an AWS-only deployment. Second, OVH is one of the few cloud providers with enough peering with AWS to stress Chop Chop without charging for egress bandwidth,

saving us from using AWS' costly bandwidth. The final cost amounted to 25,000 USD in AWS credits. Using OVH saved us more than 70,000 USD since each of Chop Chop's data point on a figure would have cost 1,700 USD in AWS egress bandwidth—21 TB at 0.08 USD per GB $\approx$ 1,700 USD.

For all experiments, we deploy one broker in each continent (Cape Town, São Paulo, Tokyo, Sydney, Frankfurt, and Northern Virginia) and one client in each of the 14 regions above, plus Tokyo and Sydney. Clients connect to their nearest broker. We configure the network for geo-distribution and high load, e.g., TCP buffer sizes [37] and UDP parameters.

All baselines run on the same parameters. For Narwhal-Bullshark, we collocate each server with one of the workers in its server group. We reproduced Narwhal-Bullshark's original experiments [69] and matched the results.

Hardware. All servers, brokers and load clients run on c6i.8xlarge machines with an Intel Xeon Platinum 8375C (32 virtual CPUs, 16 physical cores, 2.9 GHz baseline, 3.5 GHz turbo), 64 GB of memory and 12.5 Gb/s of bandwidth. We selected these machines since they provide good performance and are in the same "commodity" price range as those chosen initially for Chop Chop's main baseline: Narwhal-Bullshark. Clients run on t3.small machines: 2 vCPUs, 1 physical core, 2 GB of memory, and up to 5 Gb/s bandwidth—of which they use less than 1 KB/s. All machines run Linux Ubuntu 20.04 LTS on the AWS patched version of the Linux kernel 5.15.0, except for the load brokers on OVH which run on Linux kernel 5.4.0—the same kernel was not available.

Challenges. The most significant evaluation challenges arose from the scale of the targeted deployment. The setup and orchestration alone required simultaneous handling of up to 320 machines across two different cloud providers and 25 regions, as well as transferring 13TB of files—mostly public keys and pre-generated batches—for each of the 12 setups. To handle this, we developed a new command-line tool to efficiently deploy distributed systems: `silk`. Among other things, we use `silk` for peer-to-peer-style file transfer over aggregated TCP connections, as well as for grouped process control. With `silk`, transferring all files from a single machine takes around 30 minutes, compared to 68 hours with `scp`. The code for `silk` can be found at [*anonymized*].

Additional challenges came from the real-world nature of the targeted deployment. First, the connection between OVH and AWS's Asia and Pacific regions was particularly unstable at certain times of day especially when close to saturation. For example, Tokyo's connection was frequently degraded between 3pm and 5pm UTC. Second, the performance of some machines sometimes deviated from their specifications. As an example, in a setup size of 64, we observed around 2 machines operating with a 10% lower CPU turbo clock rate than specified. Considering these variations, we increased the number of servers a broker initially asks for witness shards (see §4.3) by a margin, e.g., $f+5$ instead of $f+1$. This improves system stability—i.e., lower latency variability—while slightly reducing maximum throughput. Unless otherwise specified, we set the margin to 4 in all experiments, i.e., $f+5$.

Plots. Every data point is the mean of 5 runs of 2 minutes each (after excluding warmup and cooldown, the relevant cross-section is at least 1 minute). All plots further depict one standard deviation from the mean using either colored shaded areas or black error bars (which may be too small to notice). Experimental data can be found at [*anonymized*].

6.3 RQ1 – Load Handling

Fig. 7 shows the latency and throughput of Chop Chop and all its baselines for various input rates of 8 B messages. The variability is represented using shaded areas.

Baselines. Both BFT-SMaRt and HotStuff showcase stable performances under low loads, respectively achieving around 1,400 and 1,600 operations per second. BFT-SMaRt's latency is consistently better than HotStuff's up to its inflection point (0.45–0.53 s vs. 1.2–1.6 s). We measure up to 3.8M op/s for Narwhal-Bullshark and up to 382k op/s for Narwhal-Bullshark-sig. The difference in respective throughputs highlights the cost of authentication for servers: verifying signatures reduces the throughput of Narwhal-Bullshark by one order of magnitude. We observe a latency of around 3.6 s for both Narwhal-Bullshark and Narwhal-Bullshark-sig.

Chop Chop. Chop Chop achieves close to 44M op/s while running on top of both HotStuff and BFT-SMaRt. Chop Chop's latency range is 3.0–3.6 s with BFT-SMaRt and 5.8–6.5 s with HotStuff. Notably, the latency of Chop Chop-HotStuff decreases under high load. This is due to the internal batching mechanism of the HotStuff implementation: buffers fill faster under higher load, thus avoiding timeouts. This has an immediate impact on Chop Chop, which feeds HotStuff at a low rate: HotStuff alone accounts for over 60% of Chop Chop-HotStuff's overall latency. BFT-SMaRt makes a better fit for Chop Chop, as its throughput is sufficient for Chop Chop's needs, and its latency is lower than HotStuff's.

Mempools' trade-off. In comparison to BFT-SMaRt and HotStuff, Chop Chop trades latency in favor of throughput. This trade-off is mostly explained by batching and distillation. When assembling a batch, a broker has to wait twice: once to collect enough messages to fill a batch, and once to collect all multi-signatures from clients engaging in distillation. We set both waits' timeout to 1 second. Notably, Narwhal-Bullshark seems to incur a similar latency cost, as Chop Chop's latency approximately matches that of Narwhal-Bullshark, even though Chop Chop needs an extra round trip between clients and broker (Fig. 5, #4–#6).

6.4 RQ2 – Distillation Benefits

We showcase the benefits of distillation by: evaluating throughput with and without distillation, evaluating distilla-

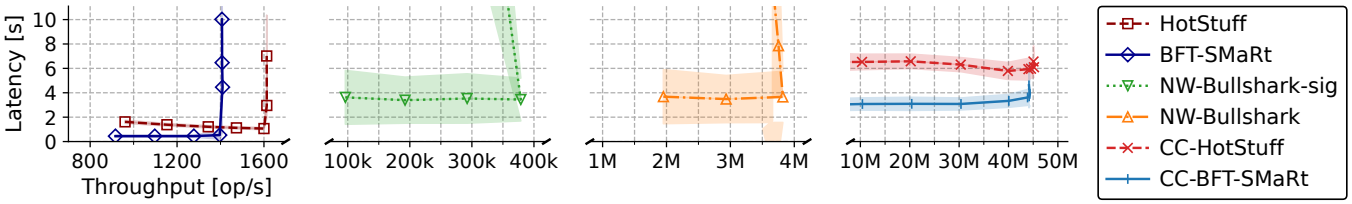


Figure 7: **Throughput-latency of Chop Chop and of notable Atomic Broadcast systems under various input rates.**

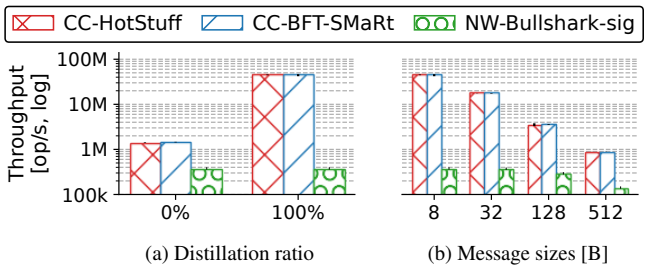


Figure 8: **Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) Chop Chop has no distillation and with (b) varying message size.**

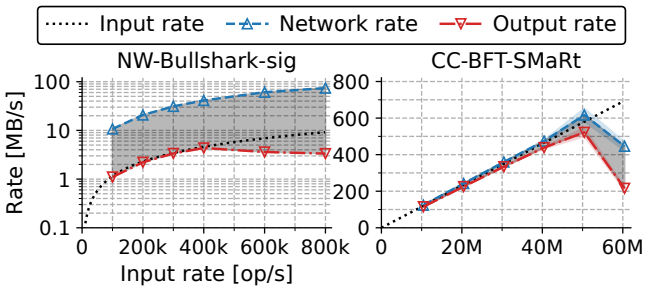


Figure 9: **Throughput efficiency of authenticated Narwhal with Bullshark (left, log scale) and Chop Chop with BFT-SMaRt (right, linear scale) with various input rates.**

tion for messages of different sizes, and observing the impact of distillation on network bandwidth to achieve line rate.

Distillation vs. mitigations. Along with distillation, Chop Chop makes use of two techniques available in the literature to mitigate the cost of Atomic Broadcast's authentication: short identifiers and pooled signature verification (see §2.2).

Fig. 8a breaks down Chop Chop's throughput, measuring how significantly distillation alone contributes to Chop Chop's performance. When no message is distilled, Chop Chop's servers bottleneck at 1.5M op/s, 3.9× higher than Narwhal-Bullshark-sig. This result is in line with both systems bottlenecking on server CPU, as the technique employed by Chop Chop to mitigate authentication complexity has only one third of the servers verify each client signature. (We conjecture that the additional 1.3 factor may be owed to engineering differences.) When batches are fully distilled, Chop Chop's throughput grows to 44M op/s, accounting for the additional 29-fold boost to Chop Chop's performance.

Distillation for larger messages. Fig. 8b illustrates Chop Chop's maximum throughput for message sizes of 8 B to 512 B which may be relevant to applications that cannot work around smaller message sizes, e.g., many smart contracts. Chop Chop's throughput is similar with BFT-SMaRt and HotStuff, decreasing at an approximately 1-to-1 ratio as the message size increases: 44.3M op/s for 8 B, 17.6M op/s for 32 B, 3.5M op/s for 128 B and 890k op/s for 512 B.

This is in line with expectations. As we discuss in §3.2, a server should receive $\sim b$ bytes in order to deliver a b-bytes message in a large, fully distilled batch, as full distillation amortizes to zero the communication cost of authenticating and sequencing each message. For 8 B messages, servers encounter a CPU bottleneck slightly before the link between load brokers and servers is saturated. This explains why the throughput decreases only 2.52× when messages grow to 32 B: all remaining server-bound bandwidth is used to convey messages (as messages are larger) while the load on server CPUs is reduced (as less messages are delivered overall). The system remains communication-bottlenecked as the size of the messages increases, and throughput starts decreasing linearly with message size, e.g., Chop Chop's throughput for 512 B messages is 4.00× smaller than for 128 B.

By contrast, Narwhal-Bullshark-sig bottlenecks on server CPUs longer, due to signature verification, maintaining a stable throughput until 512 B messages finally fill server links. Overall, Narwhal-Bullshark-sig's throughput only decreases from 382k op/s for 8 B messages to 142k op/s for 512 B messages, which matches their non-authenticated evaluation with 512 B messages. The gap between Chop Chop and Narwhal-Bullshark-sig at 512 B messages can be mostly attributed to Chop Chop's more efficient use of server bandwidth: unlike Narwhal, Chop Chop offloads the dissemination of batches to external brokers. Narwhal's use of worker-to-worker communication in its common path also makes it more prone to be affected by AWS's various upload limitations, e.g., AWS upload bandwidth is half the stated download bandwidth, and there are network credit limits for "burst" uploading.

Line rate. Fig. 9 illustrates Chop Chop's near line-rate network use by depicting its input, network and output rates:

- Input rate measures the total bytes of useful information—i.e., client identifiers and messages—that clients, load

clients and load brokers all broadcast per time unit;
- Network rate measures the ingress bandwidth of servers at their network interface, i.e., useful information captured by the input rate as well as the Atomic Broadcast's overhead for ordering, authentication and deduplication;
- Output rate, or "goodput", measures the total bytes of useful information that each server delivers per time unit.

A system with perfect line rate would match all three rates: input rate would match output rate as messages can be delivered in a timely fashion with no backlogging, and output rate would match network rate as a server would only receive useful information, with no overhead due to Atomic Broadcast. The gray-shaded areas in Fig. 9 highlight this overhead, i.e., the difference between network and input rates. Network and output rates are averaged over all servers.

In this experiment, each of the 257M simulated clients broadcast 8 B messages. This results in 11.5 B of useful information per broadcast as 28 bits = 3.5 B are sufficient to represent every identifier. This conversion is captured by the dotted line which converts the input rate from op/s, represented on the x-axis, to B/s, represented on the y-axis.

For authenticated Narwhal-Bullshark, the output rate closely matches the input rate until signature verification becomes the bottleneck at 378k op/s, shown by the plateauing output rate. The gap between Narwhal-Bullshark-sig's network and input rates is evident, differing by one order of magnitude (notably in line with our back-of-the-envelope calculation in §3.2). In contrast, thanks to distillation, Chop Chop practically achieves line-rate up to its maximum throughput. Before its inflection point at 40M op/s, the overhead of Chop Chop is less than 8%. The drop in output and network rates at 60M op/s is due to servers surpassing their computational capacity: broadcasts stall, server witness verification gets backlogged and brokers, suspecting server faults, ask for more batch witnesses, further stressing servers' CPUs.

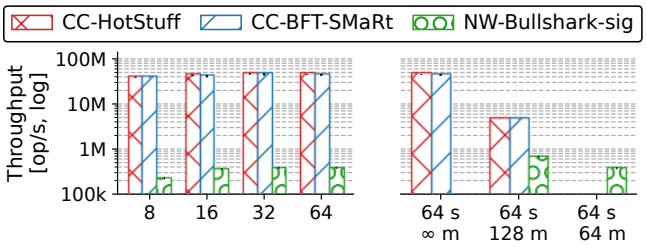
6.5 RQ3 – Number of Servers

Fig. 10a illustrates the maximum throughput for systems of 8 ($f = 2$), 16 ($f = 5$), 32 ($f = 10$) and 64 ($f = 21$) servers. For Chop Chop, we adjust the witnessing margin as the system grows by 0, 1, 2, and 4 for 8, 16, 32 and 64 servers respectively (see § 6.2). Both Chop Chop and authenticated Narwhal-Bullshark scale well to 64 servers. Note that, unless trust assumptions are modified, Narwhal-Bullshark-sig only scales vertically: if a Narwhal server or any of its workers are faulty, the entire server group is compromised. Chop Chop, instead, scales horizontally with the number of brokers.

6.6 RQ4 – Overall Efficiency

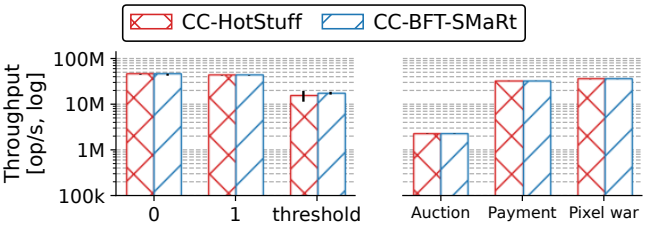
The center cluster of bars in Fig. 10b compares Chop Chop's throughput with that of authenticated Narwhal-Bullshark when overall hardware resources are matched. In this setting, both systems have 128 machines at their disposal. Chop Chop is provided with 64 servers, 64 brokers and 0 load brokers. Since a load broker uses pre-generated synthetic data to simulate tens of brokers (see §6.2), involving load brokers in this experiment would give an unfair advantage to Chop Chop. Narwhal-Bullshark-sig is provided with 128 workers, to match Chop Chop's total machines, balanced across 64 server groups, to match Chop Chop's servers. The left and right clusters of bars depict Chop Chop using load brokers and Narwhal-Bullshark-sig with 64 server groups containing 1 worker each, respectively, as in the other experiments.

We observe 4.6M op/s for Chop Chop, with servers reporting around 5% CPU usage. We observe 679k op/s for Narwhal-Bullshark-sig. Chop Chop's higher throughput is in line with expectations. In Narwhal-Bullshark-sig, workers are trusted, and as such a worker can only contribute to its own server group. Instead, since Chop Chop brokers are untrusted, a broker's work is useful to all servers.



(a) System sizes (b) Matching resources

Figure 10: **Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) varying system size, and when (b) varying the number of overall machines ("m") with 64 servers ("s").** Load brokers in Chop Chop simulate tens of brokers, hence are noted "∞ m".



(a) Server failures (b) Applications

Figure 11: **Throughput of Chop Chop (log scale) with (a) various server failures and for (b) different applications.**

6.7 RQ5 – Chop Chop Under Failures

Fig. 11a depicts Chop Chop's throughput when some servers crash 30 seconds into the run. Performance drops marginally (from 44M op/s to 43M op/s) with one crash and by 66% (down to 15M op/s) when one-third of the servers crash, resulting in less CPU globally available to witness batches.

Fig. 8a captures Chop Chop's performance hit when clients fail to engage in distillation. This could be caused by clients being slow or crashed, or brokers being malicious. Under the

most extreme conditions, where no client engages in distillation, the throughput drops from 44M op/s to 1.5M op/s.

6.8 RQ6 – Application Use Cases

Fig. 11b depicts the maximal stable throughput for various application use cases. In the Auction app, a client can bid an amount on a token it does not own, or take the highest offer it received for an item it owns. The highest amount bid on each token is locked and cannot be used to bid elsewhere. Money bid is transferred when the owner of the token takes the offer, or refunded when the bid is raised by another client. The Auction app is single-threaded and many clients bid on the same token to approximate a real auction. In the Payments app, clients choose a recipient and an amount to transfer. In Pixel war, clients choose a pixel and an RGB color to paint on a 2,048 by 2,048 board. Operations are generated at random.

We observe 2.3M op/s for the Auction, 32M op/s for Payments and 35M op/s for Pixel war. The bottleneck is the application in all cases, thus Chop Chop has sufficient capacity for high, single-application throughput. Chop Chop can also support many separate high-throughput applications simultaneously, making it a fitting Atomic Broadcast candidate to power a universal SMR system, i.e., an Internet computer.

7 Related Work

We overview below the state-of-the-art most relevant to Chop Chop, namely Atomic Broadcast systems with high-throughput and efficient signature aggregation schemes.

High-throughput Atomic Broadcast. Narwhal [26] is a mempool protocol that separates the reliable distribution of payloads from the communication-expensive ordering in order to accelerate DAG-based Atomic Broadcast [33, 42, 69]. Narwhal utilizes trusted workers to increase throughput while Chop Chop relies on *trustless* brokers, for the same effect, and scales out more efficiently. To circumvent the bottleneck associated with the broadcast leader, approaches using multiple leaders have been developed—both for crash [31, 61] and arbitrary [3, 6, 70, 71] faults—to scale the broadcast throughput linearly with the number of leaders. Dissemination trees [44, 63] have also been employed to reduce communication cost and maximize network bandwidth utility, while sharded [46, 76] and federated [54] approaches reduce communication cost by promoting local communication in geo-distributed setups. In comparison, Chop Chop shows that an optimal distillation mechanism for batches achieves better performances without adding complexity to the Atomic Broadcast protocol itself.

Other approaches have shown that the underlying hardware of servers can also be exploited for higher throughput, such as FPGA [38, 41] and Intel SGX enclaves [7]. In comparison, Chop Chop uniquely boosts throughput by exploiting *trustless hardware* via brokers. Atomic Broadcast can also be accelerated in data centers by using the topology of the network [51, 64] or even by running within the network itself using P4-programmable switches [27, 45]. In such low latency environments, the processing overhead incurred by the operating system kernel can be bypassed to further increase the throughput of Atomic Broadcast [1, 45, 75].

Signature aggregation. Aggregate signatures were first proposed to save space by compacting a large number of signatures into just one [11, 67]. Up until recently, aggregation could also save verification time but only in certain cases: either when the signatures are generated by the same signer [17, §5.1], or when the signatures are on the same message, i.e., multi-signatures [39]. In the latter case, aggregation mechanisms have been proposed to achieve constant-time verification of aggregated multi-signatures for both BLS [10] and Schnorr [57] signature schemes. In particular, multi-signatures are used in cryptocurrencies to have many servers sign the same batch of payloads [30, 44]. Servers in Chop Chop use rapidly-verifiable BLS multi-signatures [10] for that very purpose. In addition to aggregating server signatures on batches, Chop Chop's distillation mechanism also aggregates all client signatures in a batch in a way that provides constant-time verification. The theoretical scheme Draft [16] proposed signature aggregation with similar verification performances but is tailored to Reliable Broadcast. It is however unclear how Draft could be implemented as a real-world system without compromising liveness. Indeed, Draft assumes infinite memory to prevent message replay attacks which would rapidly exhaust servers' memory if deployed to match Chop Chop's maximum throughput (see §6.2). Chop Chop also aggregates client sequence numbers to significantly reduce bandwidth consumption when small messages are broadcast (Fig. 2). Chop Chop aggregates sequence numbers thanks to the ordering of and thanks to novel legitimacy proofs (see §4.2).

8 Concluding Remarks

Chop Chop's performance comes with two limitations. First, Chop Chop's high throughput makes memory management a challenge: servers fill their memory quickly if unable to garbage-collect under heavy load. Second, all servers in Chop Chop are known at startup and it is unclear if its performance would be maintained when deployed on thousands of servers. Interesting avenues of future research include sharding to achieve even higher throughput by running multiple, independent, coordinated instances of Chop Chop, and offloading more tasks to the brokers, such as public key aggregation.

Acknowledgments

We thank the OSDI '23 reviewers for their continuous involvement in the revision process. We further thank Vasileios Trigonakis for his early feedback. This work has been supported in part by AWS Cloud Credit for Research, the Hasler Foundation (#21084), and Innosuisse (46752.1 IP-ICT).

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. uBFT: Microsecond-scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[2] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS*, 102, 2010.

[3] Salem Alqahtani and Murat Demirbas. BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, 2021.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, 2018.

[5] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4), 2015.

[6] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. FnF-BFT: A BFT Protocol with Provable Performance Under Attack. In *30th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2023.

[7] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[8] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography (PKC)*, volume 3958. 2006.

[9] Alysson Bessani, Joao Sousa, and Eduardo E.P. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[10] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *Advances in Cryptology – ASIACRYPT*, 2018.

[11] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Advances in Cryptology — EUROCRYPT*, 2003.

[12] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. BLS Signatures. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-05, 2022. Work in Progress.

[13] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science, 2011.

[14] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology (JCrypt)*, 18(3), July 2005.

[15] Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. Chop Chop: Byzantine Atomic Broadcast to the Network Limit, 2024.

[16] Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira. Oracular Byzantine Reliable Broadcast. In *36th International Symposium on Distributed Computing (DISC)*, 2022.

[17] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch Verification of Short Signatures. *Journal of Cryptology (JCrypt)*, 25(4), 2012.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[19] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)*, 43(2), 1996.

[20] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[21] Authors' implementation of BFT-SMaRt in Java. https://github.com/bft-smart/library.

[22] Authors' implementation of HotStuff in C++. https://github.com/hot-stuff/libhotstuff.

[23] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[24] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Information and Computation (IC)*, 118(1), 1995.

[25] 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 *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[26] 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 Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[27] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking (ToN)*, 28(4), 2020.

[28] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4), 2004.

[29] Danny Dolev and Rüdiger Reischuk. Bounds on Information Exchange for Byzantine Agreement. *Journal of the ACM (JACM)*, 32(1), 1985.

[30] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for Consensus. In *29th USENIX Security Symposium (SEC)*, 2020.

[31] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient Replication via Timestamp Stability. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*, 2021.

[32] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2), 1985.

[33] Adam Gagol, Damian Leundefinedniak, Damian Straszak, and Michal Swietek. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT)*, 2019.

[34] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. Scaling Blockchain Consensus via a Robust Shared Mempool. In *39th IEEE International Conference on Data Engineering (ICDE)*, 2023.

[35] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[36] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[37] IBM. TCP Tuning guide. https://www.ibm.com/docs/en/linux-on-systems?topic=recommendations-network-performance-tuning.

[38] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[39] Kazuharu Itakura and Katsuhiro Nakamura. A public-key cryptosystem suitable for digital multisignatures. *NEC Research & Development*, 1983.

[40] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 https://rfc-editor.org/rfc/rfc8032, 2017.

[41] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[42] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[43] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-Fairness for Byzantine Consensus. In *Advances in Cryptology – CRYPTO*, 2020.

[44] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (SEC)*, 2016.

[45] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, 2020.

[46] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[47] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[48] Aptos Labs. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure, 2022. https://github.com/aptos-labs/aptos-core/blob/main/developer-docs-site/static/papers/whitepaper.pdf.

[49] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a State Machine. *SIGACT News*, 41(1), 2010.

[50] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

[51] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[52] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, and Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[54] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[55] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic Quorum Systems. *Information and Computation*, 170(2), 2001.

[56] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. In *2005 International Conference on Dependable Systems and Networks (DSN)*, 2005.

[57] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. *Designs, Codes and Cryptography (DCC)*, 87(9), 2019.

[58] David Mazieres. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, 2016. https://stellar.org/papers/stellar-consensus-protocol.

[59] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*, 1987.

[60] Peyman Momeni, Sergey Gorbunov, and Bohan Zhang. FairBlock: Preventing Blockchain Front-Running with Minimal Overheads. In *Security and Privacy in Communication Networks (SecureComm)*, 2023.

[61] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[62] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free Asynchronous Byzantine Consensus with T <N/3 and O(N2) Messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[63] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[64] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[65] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying Blockchain Extractable Value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[66] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[67] Claus P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology (JCrypt)*, 4(3), 1991.

[68] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT Protocols under Fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[69] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[70] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State Machine Replication Scalability Made Simple. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[71] Chrysoula Stathakopoulou, David Tudor, Matej Pavlovic, and Marko Vukolić. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research (JSys)*, 2(1), 2022.

[72] The DFINITY Team. The Internet Computer for Geeks, 2022. https://eprint.iacr.org/2022/087.

[73] The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

[74] The MystenLabs Team. The Sui Smart Contracts Platform, 2022. https://github.com/MystenLabs/sui/blob/main/doc/paper/sui.pdf.

[75] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[76] Jiaping Wang and Hao Wang. Monoxide: Scale out Blockchain with Asynchronous Consensus Zones. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[77] Roger Wattenhofer. *Blockchain Science: Distributed Ledger Technology*. Inverted Forest Publishing, 2019.

[78] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[79] 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 Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[80] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[81] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galiñanes, and Bryan Ford. Flash Freezing Flash Boys: Countering Blockchain Front-Running. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2022.

[82] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[83] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-Frequency Trading on Decentralized On-Chain Exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

A Artifact Appendix

Abstract

This artifact is a full implementation of Chop Chop, an end-to-end client-server Byzantine Atomic Broadcast system leveraging a third-party—brokers—to optimize network usage and vastly reduce the computational load of authenticating client requests on the server-side while preserving safety.

Scope

This artifact can be used to validate the following claims (*provided the setup is the same*):

- End-to-end performance (Figs. 7 and 10).
- Distillation benefits (Fig. 8) and throughput efficiency (Fig. 9).
- Performance under faults (Fig. 11).

Note that reproducing the same plots (including error bands) can be prohibitively costly given the scale of the evaluation (number and hourly price of machines, see §6.2).

Contents

The artifact contains all the source code used to implement Chop Chop. We provide a docker file to set up experiments. We also provide automated scripts to extract and interpret the data as well as generate plots. Please refer to the README.md file in the repository for more details.

Hosting

The repository can be accessed through GitHub[4] (see the instructions on the README.md file).

Requirements

There are no special hardware or software requirements beyond a recent enough version of Rust (§5) and the desired network layout to evaluate. If you wish to reproduce our results exactly, please see §6.2 for the setup used.

[4] https://github.com/Distributed-EPFL/chop-chop-osdi24