Oracular Byzantine Reliable Broadcast

Martina Camaioni
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Matteo Monti
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Byzantine Reliable Broadcast (BRB) is a fundamental distributed computing primitive, with applications ranging from notifications to asynchronous payment systems. Motivated by practical consideration, we study Client-Server Byzantine Reliable Broadcast (CSB), a multi-shot variant of BRB whose interface is split between broadcasting *clients* and delivering *servers*. We present Draft, an optimally resilient implementation of CSB. Like most implementations of BRB, Draft guarantees both liveness and safety in an asynchronous environment. Under good conditions, however, Draft achieves unparalleled efficiency. In a moment of synchrony, free from Byzantine misbehaviour, and at the limit of infinitely many broadcasting clients, a Draft server delivers a b-bits payload at an asymptotic amortized cost of 0 signature verifications, and $(log_2(c) + b)$ bits exchanged, where c is the number of clients in the system. This is the information-theoretical minimum number of bits required to convey the payload (b bits, assuming it is compressed), along with an identifier for its sender ($\log_2(c)$ bits, necessary to enumerate any set of c elements, and optimal if broadcasting frequencies are uniform or unknown). These two achievements have profound practical implications. Real-world BRB implementations are often bottlenecked either by expensive signature verifications, or by communication overhead. For Draft, instead, the network is the limit: a server can deliver payloads as quickly as it would receive them from an infallible oracle.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine reliable broadcast, Good-case complexity, Amortized complexity, Batching

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.13

Related Version The full version of this paper, which includes all detailed proofs and pseudocode, is available online.
Full Version: `https://arxiv.org/abs/` [17]

1 Introduction

Byzantine reliable broadcast (BRB) is one of the most fundamental and versatile building blocks in distributed computing, powering a variety of Byzantine fault-tolerant (BFT) systems [14, 28]. The BRB abstraction has recently been shown to be strong enough to process payments, enabling cryptocurrency deployments in an asynchronous environment [29]. Originally introduced by Bracha [9] to allow a set of processes to agree on a single message from a designated sender, BRB naturally generalizes to the multi-shot case, enabling higher-level abstractions such as Byzantine FIFO [44, 12] and causal [7, 4] broadcast. We study a practical, multi-shot variant of BRB whose interface is split between broadcasting *clients* and delivering *servers*. We call this abstraction Client-Server Byzantine Reliable Broadcast (CSB).

© Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 13; pp. 13:1–13:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

CSB in brief. Clients broadcast, and servers deliver, *payloads* composed by a *context* and a *message*. This interface allows, for example, Alice to announce her wedding as well as will her fortune by respectively broadcasting

$$\Bigg(\underbrace{\texttt{"My wife is"}}_{\text{context } c_w}, \underbrace{\texttt{"Carla"}}_{\text{message } m_w}\Bigg) \quad \Bigg(\underbrace{\texttt{"All my riches go to"}}_{\text{context } c_r}, \underbrace{\texttt{"Bob"}}_{\text{message } m_r}\Bigg)$$

CSB guarantees that: (*Consistency*) no two correct servers deliver different messages for the same client and context; (*Totality*) either all correct servers deliver a message for a given client and context, or no correct server does; (*Integrity*) if a correct server delivers a payload from a correct client, then the client has broadcast that payload; and (*Validity*) a payload broadcast by a correct client is delivered by at least one correct server. Following from the above example, Carla being Alice's wife does not conflict with Bob being her sole heir (indeed, $c_w \neq c_r$), but Alice would not be able to convince two correct servers that she married Carla and Diana, respectively. Higher-level broadcast abstractions can be easily built on top of CSB. For example, using integer sequence numbers as contexts and adding a reordering layer yields Client-Server Byzantine FIFO Broadcast. For the sake of CSB, however, it is not important for contexts to be integers, or satisfy any property other than comparability. Throughout the remainder of this paper, the reader can picture contexts as opaque binary blobs. Lastly, while the set of servers is known, CSB as presented does not assume any client to be known a priori. The set of clients can be permissionless, with servers discovering new clients throughout the execution.

A utopian model. Real-world BRB implementations are often bottlenecked either by expensive signature verifications [21] or by communication overhead [10, 34, 35]. With the goal of broadening those bottlenecks, simplified, more trustful models are useful to establish a (sometimes grossly unreachable) bound on the efficiency that an algorithm can attain in the Byzantine setting. For example, in a utopian model where any agreed-upon process can be trusted to never fail (let us call it an *oracle*), CSB can easily be implemented with great efficiency. Upon initialization, the oracle organizes all clients in a list, which it disseminates to all servers. For simplicity, let us call *id* a client's position in the list. To broadcast a payload p, a client with id i simply sends p to the oracle: the oracle checks p for equivocation (thus ensuring consistency), then forwards (i, p) to all servers (thus ensuring validity and totality). Upon receiving (i, p), a server blindly trusts the oracle to uphold all CSB properties, and delivers (i, p). Oracle-CSB is clearly very efficient. On the one hand, because the oracle can be trusted not to attribute spurious payloads to correct clients, integrity can be guaranteed without any server-side signature verification. On the other, in order to deliver (i, p), a server needs to receive just $(\lceil \log_2(c) \rceil + |p|)$ bits, where c denotes the total number of clients, and $|p|$ measures p's length in bits. This is optimal assuming the rate at which clients broadcast is unknown[1] or uniform[2] [20].

Matching the oracle. Due to its reliance on a single infallible process, Oracle-CSB is not a fault-tolerant distributed algorithm: shifting back to the Byzantine setting, a single failure would be sufficient to compromise all CSB properties. Common sense suggests that Byzantine

[1] Lacking an assumption on broadcasting rates, an adversarial scheduler could have all messages broadcast by the client with the longest id, which we cannot guarantee to be shorter than $\lceil \log_2(c) \rceil$ bits.

[2] Should some clients be expected to broadcast more frequently than others, we could further optimize Oracle-CSB by assigning smaller ids to more active clients, possibly at the cost of having less active clients have ids whose length exceeds $\lceil \log_2(c) \rceil$. Doing so, however, is beyond the scope of this paper.

resilience will necessarily come at some cost: protocol messages must be exchanged to preserve consistency and totality, signatures must be produced and verified to uphold integrity and, lacking the totally-ordering power that only consensus can provide, ids cannot be assigned in an optimally dense way. However, this paper proves the counter-intuitive result that an asynchronous, optimally-resilient, Byzantine implementation of CSB can asymptotically match the efficiency of Oracle-CSB. This is not just up to a constant, but identically. In a synchronous execution, free from Byzantine misbehaviour, and as the number of concurrently broadcasting clients goes to infinity (we call these conditions the *batching limit*[3]), our CSB implementation Draft delivers a payload p at an asymptotic[4], amortized cost of 0 signature verifications[5] and $(\lceil \log_2(c) \rceil + |p|)$ bits exchanged per server, the same as in Oracle-CSB (we say that Draft achieves *oracular efficiency*). At the batching limit a Draft server is dispensed from nearly all signature verifications, as well as nearly all traffic that would be normally required to convey protocol messages, signatures, or client public keys. Network is the limit: payloads are delivered as quickly as they can be received.

CSB's common bottlenecks. To achieve oracular efficiency, we focus on three types of server overhead that commonly affect a real-world implementation of CSB:

- *Protocol overhead*. Safekeeping consistency and totality typically requires some form of communication among servers. This communication can be direct (as in Bracha's original, all-to-all BRB implementation) or happen through an intermediary (as in Bracha's signed, one-to-all-to-one BRB variant), usually employing signatures to establish authenticated, intra-server communication channels through a (potentially Byzantine) relay.
- *Signature overhead*. Upholding integrity usually requires clients to authenticate their messages using signatures. For servers, this entails both a computation and a communication overhead. On the one hand, even using well-optimized schemes, signature verification is often CPU-heavy enough to dominate a server's computational budget, dwarfing in particular the CPU footprint of much lighter, symmetric cryptographic primitives such as hashes and ciphers. On the other hand, transmitting signatures results in a fixed communication overhead per payload delivered. While the size of a signature usually ranges from a few tens to a few hundreds of bytes, this overhead is non-negligible in a context where many clients broadcast small messages. This is especially true in the case of payments, where a message reduces to the identifier of a target account and an integer to denote the amount of money to transfer.
- *Identifier overhead*. CSB's multi-shot nature calls for a sender identifier to be attached to each broadcast payload. Classically, the client's public key is used as identifier. This is convenient for two reasons. First, knowing a client's identifier is sufficient to authenticate its payloads. Second, asymmetric keypairs have very low probability of collision. As such, clients can create identities in the system without any need for coordination: locally generating a keypair is sufficient to begin broadcasting messages. By cryptographic design, however, public keys are sparse, and their size does not change with the number of clients. This translates to tens to hundreds of bytes being invested to identify a client from a set that can realistically be enumerated by a few tens of bits. Again, this communication overhead is heavier on systems where broadcasts are frequent and brief.

[3] The batching limit includes other easily achievable, more technical conditions that we omit in this section for the sake of brevity. For the full definition, please refer to the extended version of this paper [17].
[4] The asymptotic costs are reached quite fast, at rates comparable to C^{-1} or $log(C) \cdot C^{-1}$.
[5] This does not mean that batches are processed in constant time: hashes and signature aggregations, for example, still scale linearly in the size of a batch. The real-world computational cost of such simple operations, however, is several orders of magnitude lower than that of signature verification.

On the way to matching Oracle-CSB's performance, we develop techniques to negate all three types of overhead: at the batching limit, a Draft server delivers a payload wasting 0 bits to protocol overhead, performing 0 signature verifications, and exchanging $\lceil \log_2(c) \rceil$ bits of identifier, the minimum required to enumerate the set of clients. We outline our contributions below, organized in three (plus one) take-home messages (T-HMs).

T-HM1: The effectiveness of batching goes beyond total order. In the totally ordered setting, batching is famously effective at amortizing protocol overhead [45, 3]. Instead of disseminating its message to all servers, a client hands it over to (one or more)[6] batching processes. Upon collecting a large enough set of messages, a batching process organizes all messages in a batch, which it then disseminates to the servers. Having done so, the batching process submits the batch's hash to the system's totally-ordering primitive. Because hashes are constant in length, the cost of totally ordering a batch does not depend on its size. Once batches are totally ordered, so too are messages (messages within a batch can be ordered by any deterministic function), and equivocations can be handled at the application layer (for example, in the context of a cryptocurrency, the second request to transfer the same asset can be ignored by all correct servers, with no need for additional coordination). At the limit of infinitely large batches, the relative overhead of the ordering protocol becomes vanishingly small, and a server can allocate virtually all of its bandwidth to receiving batches. This strategy, however, does not naturally generalize to CSB, where batches lack total order. As payloads from multiple clients are bundled in the same batch, a correct server might detect equivocation for only a subset of the payloads in the batch. Entirely accepting or entirely rejecting a partially equivocated batch is not an option. In the first case, consistency could be violated. In the second case, a single Byzantine client could single-handedly "poison" the batches assembled by every correct batching process with equivocated payloads, thus violating validity. In Draft, a server can partially reject a batch, acknowledging all but some of its payloads. Along with its partial acknowledgement, a server provides a proof of equivocation to justify each exception. Having collected a quorum of appropriately justified partial acknowledgements, a batching process has servers deliver only those payloads that were not excepted by any server. Because proofs of equivocations cannot be forged for correct clients, a correct client handing over its payload to a correct batching process is guaranteed to have that payload delivered. In the common case where batches have little to no equivocations, servers exchange either empty or small lists of exceptions, whose size does not scale with that of the batch. This extends the protocol-amortizing power of batching to CSB and, we conjecture, other non-totally ordered abstractions.

T-HM2: Interactive multi-signing can slash signature overhead. Traditionally, batching protocols are non-interactive on the side of clients. Having offloaded its message to a correct batching process, a correct client does not need to interact further for its message to be delivered: the batching process collects an arbitrary set of independently signed messages and turns to the servers to get each signature verified, and the batch delivered. This approach is versatile (messages are not tied to the batch they belong to) and reliable (a client crashing does not affect a batch's progress) but expensive (the cost of verifying each signature is high and independent of the batch's size). In Draft, batching processes engage in an interactive protocol with clients to replace, in the good case, all individual signatures in a batch with

[6] In most real-world implementations, a client optimistically entrusts its payload to a single process, extending its request to larger portions of the system upon expiration of a suitable timeout.

a single, batch-wide *multi-signature*. In brief, multi-signature schemes extend traditional signatures with a mechanism to *aggregate* signatures and public keys: an arbitrarily large set of signatures for the same message[7] can be aggregated into a single, constant-sized signature; similarly, a set of public keys can be aggregated into a single, constant-sized public key. The aggregation of a set of signatures can be verified in constant time against the aggregation of all corresponding public keys. Unlike verification, aggregation is a cheap operation, reducing in some schemes to a single multiplication on a suitable field. Multi-signature schemes open a possibility to turn expensive signature verification into a once-per-batch operation. Intuitively, if each client contributing to a batch could multi-sign the entire batch instead of its individual payload, all multi-signatures could be aggregated, allowing servers to authenticate all payloads at once. However, as clients cannot predict how their payloads will be batched, this must be achieved by means of an interactive protocol. Having collected a set of individually-signed payloads in a batch, a Draft batching process shows to each contributing client that its payload was included in the batch. In response, clients produce their multi-signatures for the batch's hash, which the batching process aggregates. Clients that fail to engage in this interactive protocol (e.g., because they are faulty or slow) do not lose liveness, as their original signature can still be attached to the batch to authenticate their individual payload. In the good case, all clients reply in a timely fashion, and each server has to verify a single multi-signature per batch. At the limit of infinitely large batches, this results in each payload being delivered at an amortized cost of 0 signature verifications. The usefulness of this interactive protocol naturally extends beyond CSB to all multi-shot broadcast abstractions whose properties include integrity.

T-HM3: Dense id assignment can be achieved without consensus. In order to efficiently convey payload senders, Oracle-CSB's oracle organizes all clients in a list, attaching to each client a successive integral identifier. Once the list is disseminated to all servers, the oracle can identify each client by its identifier, sparing servers the cost of receiving larger, more sparse, client-generated public keys. Id-assignment strategies similar to that of Oracle-CSB can be developed, in the distributed setting, building on top of classical algorithms that identify clients by their full public keys (we call such algorithms *id-free*, as opposed to algorithms such as Draft, which are *id-optimized*). In a setting where consensus can be achieved, the identifier density of Oracle-CSB is easily matched. Upon initialization, each client submits its public key to an id-free implementation of Total-Order Broadcast (TOB). Upon delivery of a public key, every correct process agrees on its position within the common, totally-ordered log. As in Oracle-CSB, each client can then use its position in the list as identifier within some faster, id-optimized broadcast implementation. In a consensus-less setting, achieving a totally-ordered list of public keys is famously impossible [26]. This paper, however, proves the counter-intuitive result that, when batching is used, the density of ids assigned by a consensus-less abstraction can asymptotically match that of those produced by Oracle-CSB or consensus. In Dibs, our consensus-less id-assigning algorithm, a client requests an id from every server. Each server uses an id-free implementation of FIFO Broadcast to order the client's public key within its own log. Having observed its public key appear in at least one log, the client publicly elects the server in charge of that log to be its *assigner*. Having done so, the client obtains an id composed of the assigner's public key and the client's position

[7] Some multi-signature schemes also allow the aggregation of signatures on heterogeneous messages. In that case, however, aggregation is usually as expensive as signature verification. Given our goal to reduce CPU complexity for servers, this paper entirely disregards heterogeneous aggregation schemes.

within the assigner's log. We call the two components of an id *domain* and *index*, respectively. Because the set of servers is known to (and can be enumerated by) all processes, an id's domain can be represented in $\lceil \log_2(n) \rceil$ bits, where n denotes the total number of servers. Because at most c distinct clients can appear in the FIFO log of any server, indices are at most $\lceil \log_2(c) \rceil$ bits long. In summary, Dibs assigns ids to clients without consensus, at an additional cost of $\lceil \log_2(n) \rceil$ bits per id. Interestingly, even this additional complexity can be amortized by batching. Having assembled a batch, a Draft batching process represents senders not as a list of ids, but as a map, associating to each of the n domains the indices of all ids in the batch under that domain. At the limit of infinitely large batches ($C \gg N$), the bits required to represent the map's keys are entirely amortized by those required to represent its values. This means that, while $(\lceil \log_2(n) \rceil + \lceil \log_2(c) \rceil)$ bits are required to identify a client in isolation, $\lceil \log_2(c) \rceil$ bits are sufficient if the client is batched: even without consensus, Draft asymptotically matches the id efficiency of Oracle-CSB.

Bonus T-HM: Untrusted processes can carry the system. In THM1, we outlined how batching can be generalized to the consensus-less case, and discussed its role in removing protocol overhead. In THM2, we sketched how an interactive protocol between clients and batching processes can eliminate signature overhead. In employing these techniques, we shifted most of the communication and computation complexity of our algorithms from servers to batching processes. Batching processes verify all client signatures, create batches, verify and aggregate all client multi-signatures, then communicate with servers in an expensive one-to-all pattern, engaging server resources (at the batching limit) as little as an oracle would. Our last contribution is to observe that a batching process plays no role in upholding CSB's safety. As we discuss in detail throughout the remainder of this paper, a malicious batching process cannot compromise consistency (it would need to collect two conflicting quorums of acknowledgements), totality (any server delivering a batch has enough information to convince all others to do the same) or integrity (batches are still signed, and forged or improperly aggregated multi-signatures are guaranteed to be detected). Intuitively, the only damage a batching process can do to the system is to refuse to process client payloads[8]. This means that a batching process does not need to satisfy the same security properties as a server. CSB's properties cannot be upheld if a third of the servers are faulty. Conversely, Draft has both liveness and safety as long as *a single* batching process is correct. This observation has profound practical implications. In the real world, scaling the resources of a permissioned, security-critical set of servers can be hard. On the one hand, reputable, dependable institutions partaking in the system might not have the resources to keep up with its demands. On the other, more trusted hardware translates to a larger security cross-section. Trustless processes, however, are plentiful to the point that permissionless cryptocurrencies traditionally waste their resources, making them compete against each other in expensive proofs of Sybil-resistance [39]. In this paper, we extend the classical client-server model with *brokers*, a permissionless, scalable set of processes whose only purpose is to alleviate server complexity. Unlike servers, more than two-thirds of which we assume to be correct, all brokers but one can be faulty. In Draft, brokers act as an intermediary between clients and servers, taking upon themselves the batching of payloads, verification and aggregation of signatures, the dissemination of batches, and the transmission of protocol messages.

[8] Or cause servers to waste resources, e.g., by transmitting improperly signed batches. Simple accountability measures, we conjecture, would be sufficient to mitigate these attacks in Draft. A full discussion of Denial of Service, however, is beyond the scope of this paper.

Roadmap. We discuss related work in Section 2. We state our model and recall useful cryptographic background in Section 3. In Section 4, we introduce our CSB implementation Draft: we overview Draft's protocol in Section 4.1, and provide high-level arguments for Draft's efficiency in Section 4.2. We draw our conclusions and propose future work in Section 5. The full formal analysis of our algorithms as well as their pseudocode can be found in the extended version of this paper [17].

2 Related Work

Byzantine Reliable Broadcast (BRB) is a classical primitive of distributed computing, with widespread practical applications such as in State Machine Replication (SMR) [38, 15, 11], Byzantine agreement [40, 18, 32, 31, 47], blockchains [3, 22, 23], and online payments [29, 19, 33]. In classical BRB, a system of n processes agree on a single message from a single *source* (one of the n processes), while tolerating up to f Byzantine failures (f of the n processes can behave arbitrarily). A well known solution to asynchronous BRB with provably optimal resilience ($f < n/3$) was first proposed by Bracha [8, 9] who introduced the problem. Bracha's broadcast reaches $O(n^2)$ message complexity, and $O(n^2L)$ communication complexity (total number of transmitted bits between correct processes [48]), where L is the length of the message. Since $O(n^2)$ message complexity is provably optimal [27], the main focus of BRB-related research has been on reducing its communication complexity. The best lower bound for communication complexity is $\Omega(nL + n^2)$, although it is unknown whether it is tight. The nL term comes from all processes having to receive the message (length L), while the n^2 term comes from each of the n processes having to receive $\Omega(n)$ protocol messages to ensure agreement in the presence of $f = \Theta(n)$ failures [27]. One line of research focuses on worst-case complexity, predominantly using error correcting codes [43, 6] or erasure codes [41, 30, 16, 2], and has produced various BRB protocols with improved complexity [2, 16, 13, 24, 40], many of them quite recently. The work of Das, Xiang and Ren [24] achieves $O(nL + kn^2)$ communication complexity (specifically, $7nL + 2kn^2$), where k is the security parameter (e.g., the length of a hash, typically 256 bits). As the authors note, the value of hidden constants (and k, which is sometimes considered as a constant in literature) is particularly important when considering practical implementations of these protocols. Another line of research focuses on optimizing the good case performance of BRB, i.e., when the network behaves synchronously and no process misbehaves [13, 18, 32, 42, 1]. As the good case is usually the common case, in practice, the real-world communication complexity of these optimistic protocols matches that of the good case. A simple and widely-used hash-based BRB protocol is given by Cachin *et al.* [13]. It replaces the echo and ready phase messages in Bracha's protocol with hashes, achieving $O(nL + kn^2)$ in the good case (specifically, $nL + 2kn^2$), and $O(n^2L)$ in the worst-case. Considering practical throughput, some protocols also focus on the *amortized* complexity per source message [18, 42, 36]. Combining techniques such as *batching* [18] and threshold signatures [46], at the limit (of batch size), BRB protocols reach $O(nL)$ amortized communication complexity in the good case [42]. At this point, the remaining problem lies in the hidden constants. In the authenticated setting, batching-based protocols rely on digital signatures to validate (source) messages before agreeing to deliver them [42]. In reality, each source message in a batch includes its content, an identifier of the source (e.g., a k-sized public key), a sequence id (identifying the message), and a k-sized signature. When considering systems where L is small (e.g., online payments), these can take up a large fraction of the communication. To be precise, the good-case amortized communication complexity would be $O(nL + kn)$. In fact, message signatures (the kn factor)

are by far the main bottleneck in practical applications of BRB today [23, 47], both in terms of communication and computation (signature verification), leading to various attempts at reducing or amortizing their cost [22, 36]. For example, Crain *et al.* [22] propose *verification sharding*, in which only $f+1$ processes have to receive and verify all message signatures in the good case, which is a 3-fold improvement over previous systems (on the kn factor) where all n processes verify all signatures. However, by itself, this does not improve on the amortized cost of $O(nL+kn)$ per message. When contrasting theoretical research with practical systems, it is interesting to note the gap that can surge between the theoretical model and reality. The recent work of Abraham *et al.* [1], focused on the good-case latency of Byzantine broadcast, expands on some of these mismatches and argues about the practical limitations of focusing on the worst-case. Another apparent mismatch lies in the classical model of Byzantine broadcast. In many of the applications of BRB mentioned previously (e.g., SMR, permissioned blockchains, online payments), there is usually a set of *servers* (n, up to f of which are faulty), and a set of external clients (X) which are the true sources of messages. The usual transformation from BRB's classical model into these practical settings maps the set of n servers as the n processes and simply excludes clients as system entities, e.g., assuming their messages are relayed through one of the servers. Since the number of clients can be very large ($|X| \gg n$), clients are untrusted (which can limit their usefulness), and the focus is on the communication complexity of the *servers*, this transformation seems reasonable and simplifies the problem. However, it can also limit the search for more practical solutions. In this paper, in contrast with the classical model of BRB, we explicitly include the set of clients X in our system while focusing on the communication complexity surrounding the servers (i.e., the bottleneck). Furthermore, we introduce *brokers*, an untrusted set B of processes, only one of which is assumed to be correct, whose goal is to assist servers in their operation. By doing this, we can leverage brokers to achieve a good-case, amortized communication complexity (for servers, information received or sent) of $nL+o(nL)$.

3 Model & background

3.1 Model

System and adversary. We assume an asynchronous message-passing system where the set Π of processes is the distinct union of three sets: **servers** (Σ), **brokers** (B), and **clients** (X). We use $n=|\Sigma|$, $k=|B|$ and $c=|X|$. Any two processes can communicate via reliable, FIFO, point-to-point links (messages are delivered in the order they are sent). Faulty processes are Byzantine, i.e., they may fail arbitrarily. Byzantine processes know each other, and may collude and coordinate their actions. At most f servers are Byzantine, with $n=3f+1$. At least one broker is correct. All clients may be faulty. We use Π_C and Π_F to respectively identify the set of correct and faulty processes. The adversary cannot subvert cryptographic primitives (e.g., forge signatures). Servers and brokers[9] are permissioned (every process knows Σ and B), clients are permissionless (no correct process knows X a priori). We call *certificate* a statement signed by either a plurality ($f+1$) or a quorum ($2f+1$) of servers. Since every process knows Σ, any process can verify a certificate.

[9] The assumption that brokers are permissioned is made for simplicity, and can be easily relaxed to the requirement that every correct process knows at least one correct broker.

Good case. The algorithms presented in this paper are designed to uphold all their properties in the model above. Draft, however, achieves oracular efficiency only in the *good case*. In the good case, links are synchronous (messages are delivered at most one time unit after they are sent), all processes are correct, and the set of brokers contains only one element. To take advantage of the good case, Draft makes use of timers (which is uncommon for purely asynchronous algorithms). A timer with timeout δ set at time t rings: after time $(t + \delta)$, if the system is synchronous; after time t, otherwise. Intuitively, in the non-synchronous case, timers disregard their timeout entirely, and are guaranteed to ring only eventually.

3.2 Background

Besides commonly used hashes and signatures, the algorithms presented in this paper make use of two less often used cryptographic primitives, namely, multi-signatures and Merkle trees. We briefly outline their use below. An in-depth discussion of their inner workings, however is beyond the scope of this paper.

Multi-signatures. Like traditional signatures, multi-signatures [5] are used to publicly authenticate messages: a public / secret keypair (p, r) is generated locally; r is used to produce a signature s for a message m; s is publicly verified against p and m. Unlike traditional signatures, however, multi-signatures for the same message can be *aggregated*. Let $(p_1, r_1), \ldots, (p_n, r_n)$ be a set of keypairs, let m be a message, and let s_i be r_i's signature for m. $(p_1, \ldots, p_n)$ and $(s_1, \ldots, s_n)$ can be respectively aggregated into a constant-sized public key $\hat{p}$ and a constant-sized signature $\hat{s}$. As with individually-generated multi-signatures, $\hat{s}$ can be verified in constant time against $\hat{p}$ and m. Aggregation is cheap and non-interactive: provided with $(p_1, \ldots, p_n)$ (resp., $(s_1, \ldots, s_n)$) any process can compute $\hat{p}$ (resp., $\hat{s}$).

Merkle trees. Merkle trees [37] extend traditional hashes with compact *proofs of inclusion*. As with hashes, a sequence $(x_1, \ldots, x_n)$ of values can be hashed into a preimage and collision-resistant digest (or *root*) r. Unlike hashes, however, a proof p_i can be produced from $(x_1, \ldots, x_n)$ to attest that the i-th element of the sequence whose root is r is indeed x_i. In other words, provided with r, p_i and x_i, any process can verify that the i-th element of $(x_1, \ldots, x_n)$ is indeed x_i, without having to learn $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. The size of a proof of inclusion for a sequence of n elements is logarithmic in n.

4 Draft: Overview

In this section, we provide an intuitive overview of our CSB implementation, Draft, as well as high-level arguments for its efficiency.

4.1 Protocol

Dramatis personae. The goal of this section is to provide an intuitive understanding of Draft's protocol. In order to do this, we focus on four processes: a correct client χ, a correct broker β, a correct and fast server σ, and a correct but slow server $\tilde{\sigma}$. We follow the messages exchanged between χ, β, σ and $\tilde{\sigma}$ as the protocol unfolds, as captured by Figure 1.

The setting. χ's goal is to broadcast a payload p. χ has already used Draft's underlying Directory abstraction (DIR) to obtain an id i. In brief, DIR guarantees that i is assigned to χ only, and provides χ with an *assignment certificate* a, which χ can use to prove that its id

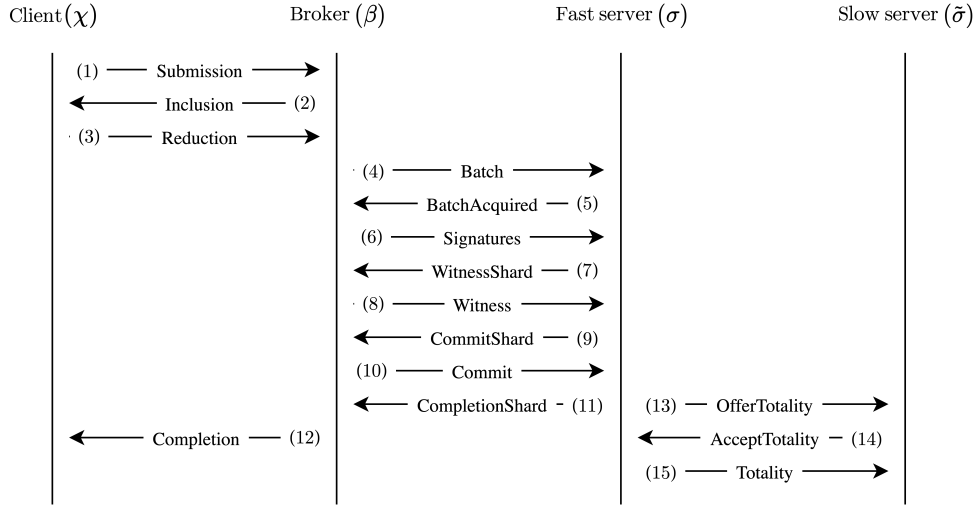

Figure 1 Draft's protocol. Having collected a batch of client payloads, a broker engages in an interactive protocol with clients to reduce the batch, replacing (most of) its individual payload signatures with a single, batch-wide multi-signature. The broker then disseminates the batch to all servers, successively gathering a witness for its correctness and a certificate to commit (some of) its payloads. Having had a plurality of servers deliver the batch, the broker notifies all clients with a suitable certificate. In the bad case, servers can ensure totality without any help from the broker, propagating batches and commit certificates in an all-to-all fashion.

is indeed i. As we discussed in Section 1, Draft uses DIR-assigned ids to identify payload senders. This is essential to Draft's performance, as DIR guarantees *density*: as we outline in Section 4.2, $\lceil \log_2 (c) \rceil$ bits are asymptotically sufficient to represent each id in an infinitely large batch. Throughout the remainder of this paper, we say that a process π *knows* an id $\hat{i}$ iff π knows the public keys to which $\hat{i}$ is assigned.

Building a batch. In order to broadcast its payload p, χ produces a signature s for p, and then sends a `Submission` message to β (fig. 1, step 1). The `Submission` message contains p, s, and χ's assignment certificate a. Upon receiving the `Submission` message, β learns χ's id i from a, then verifies s against p. Having done so, β stores (i, p, s) in its *submission pool*. For a configurable amount of time, β fills its pool with submissions from other clients, before *flushing* it into a *batch*. Let us use $(i_1, p_1, s_1), \ldots, (i_b, p_b, s_b)$ to enumerate the elements β flushes from the submission pool (for some n, we clearly have $(i, p, s) = (i_n, p_n, s_n)$). For convenience, we will also use χ_j to identify the sender of p_j (owner of i_j). Importantly, β flushes the pool in such a way that $i_j \neq i_k$ for all $j \neq k$: for safety reasons that will soon be clear, Draft's protocol prevents a client from having more than one payload in any specific batch. Because of this constraint, some payloads might linger in β's pool. This is not an issue: β will simply flush those payloads to a different batch at a later time. When building the batch, β splits submissions and signatures, storing $(i_1, p_1), \ldots, (i_b, p_b)$ separately from $s_1, \ldots, s_b$.

Reducing the batch. Having flushed submissions $(i_1, p_1), \ldots, (i_b, p_b)$ and signatures $s_1, \ldots, s_b$, β moves on to *reduce* the batch, as exemplified in Figure 2. In an attempt to minimize signature overhead for servers, β engages in an interactive protocol with clients $\chi_1, \ldots, \chi_b$ to replace as many signatures as possible with a single, batch-wide multi-signature.

In order to do so, β organizes $(i_1, p_1), \ldots, (i_b, p_b)$ in a Merkle tree with root r (for brevity, we call r the batch's root). β then sends an `Inclusion` message to each χ_j (fig. 1, step 2). Each `Inclusion` message contains r, along with a proof of inclusion q_j for (i_j, p_j). Upon receiving its `Inclusion` message, χ checks q_n against r. In doing so, χ comes to two conclusions. First, χ's submission $(i, p) = (i_n, p_n)$ is part of a batch whose root is r. Second, because no Draft batch can contain multiple payloads from the same client, that batch does not attribute χ any payload other than p. In other words, χ can be certain that β will not broadcast some spurious payload $p' \neq p$ in χ's name: should β attempt to do that, the batch would be verifiably malformed, and immediately discarded. This means χ can safely produce a multi-signature m for r: as far as χ is concerned, the batch with root r upholds integrity. Having signed r, χ sends m to β by means of a `Reduction` message (fig. 1, step 3). Upon receiving χ_j's `Reduction` message, β checks χ_j's multi-signature m_j against r. Having done so, β discards χ_j's original signature s_j. Intuitively, with m_j, χ_j attested its agreement with whatever payload the batch attributes to χ_j. Because this is equivalent to individually authenticating p_j, s_j is redundant and can be dropped. Upon expiration of a suitable timeout, β stops collecting `Reduction` messages: clearly, if β waited for every χ_j to produce m_j, a single Byzantine client could prevent the protocol from moving forward by refusing to send its `Reduction` message. β aggregates all the multi-signatures it collected for r into a single, batch-wide multi-signature m. In the good case, every χ_j is correct and timely. If so, β drops all individual signatures, and the entire batch is authenticated by m alone.

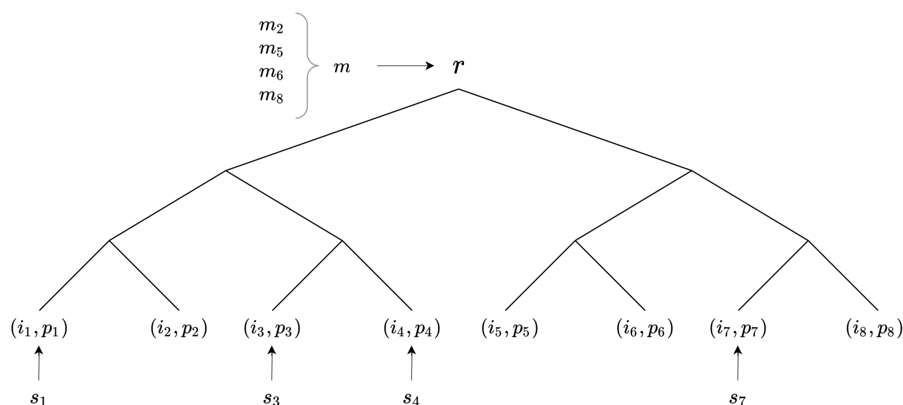

Figure 2 An example of partially reduced batch. $B = 8$ submissions are organized on the leaves of a Merkle tree with root r. Each submission (i_j, p_j) is originally authenticated by an individual signature s_j. Upon collecting a multi-signature m_j for r, the broker drops s_j. Here the broker collected multi-signatures m_2, m_5, m_6 and m_8, leaving a *straggler set* $S = \{(i_1, s_1), (i_3, s_3), (i_4, s_4), (i_7, s_7)\}$. Upon expiration of a suitable timeout, the broker aggregates m_2, m_5, m_6 and m_8 into a single multi-signature m. As such, every payload in the batch is authenticated either by m or by S.

The perks of a reduced batch. Having reduced the batch, β is left with a sequence of submissions $(i_1, p_1), \ldots, (i_b, p_b)$, a multisignature m on the Merkle root r of $(i_1, p_1), \ldots, (i_b, p_b)$, and a *straggler* set S holding the individual signatures that β failed to reduce. More precisely, S contains (i_j, s_j) iff β did not receive a valid `Reduction` message from χ_j before the reduction timeout expired. We recall that m's size is constant, and S is empty in the good case. Once reduced, the batch is cheap to authenticate: it is sufficient to verify the batch's

multi-signature against the batch's root, and each straggler signature against its individual payload. More precisely, let T denote the set of *timely* clients (χ_j is in T iff $(i_j, ..)$ is not in S). Let t denote the aggregation of T's public keys. Provided with $(i_1, p_1), \ldots, (i_b, p_b)$, m and S, any process that knows $i_1, \ldots, i_b$ can verify that the batch upholds integrity by: (1) computing r and t from $(i_1, p_1), \ldots, (i_b, p_b)$ and S; (2) using t to verify m against r; and (3) verifying each s_j in S against p_j. In the good case, authenticating the batch reduces to verifying a single multi-signature. This is regardless of the batch's size.

The pitfalls of a reduced batch. As we discussed in the previous paragraph, reducing a batch makes it cheaper to verify its integrity. Reduction, however, hides a subtle trade-off: once reduced, a batch gets easier to authenticate *as whole*. Its *individual payloads*, however, become harder to authenticate. For the sake of simplicity, let us imagine that β successfully dropped all the individual signatures it originally gathered from $\chi_1, \ldots, \chi_b$. In order to prove that $(\chi = \chi_n)$ broadcast $(p = p_n)$, β could naively produce the batch's root r, (i_n, p_n)'s proof of inclusion q_n, and the batch's multi-signature m for r. This, however, would not be sufficient to authenticate p: because the multi-signature m_n that χ produced for r was aggregated with all others, m can only be verified by the aggregation of *all* $\chi_1, \ldots, \chi_b$'s public keys. This makes authenticating p as expensive as authenticating the entire batch: in order to verify m, *all* (i_j, p_j) must be produced and checked against r, so that all corresponding public keys can be safely aggregated.

Witnessing the batch. As we discuss next, proving the integrity of individual payloads is fundamental to ensure Draft's validity. In brief, to prove that some χ_k equivocated its payload $p_k = (c_k, l_k)$, a server must prove to β that χ_k also issued some payload $p'_k = (c_k, l'_k \neq l_k)$. Lacking this proof, a single Byzantine server could, for example, claim without basis that χ equivocated p. This could trick β into excluding p, thus compromising Draft's validity. As we discussed in the previous paragraph, however, proving the integrity of an individual payload in a reduced batch is difficult. While we conjecture that purely cryptographic solutions to this impasse might be achievable in some schemes[10], Draft has β engage in a simple protocol to further simplify the batch's authentication, replacing all client-issued (multi-)signatures with a single, server-issued certificate. Having collected and reduced the batch, β sends a `Batch` message to all servers (fig. 1, step 4). The `Batch` message only contains $(i_1, p_1), \ldots, (i_b, p_b)$. Upon receiving the `Batch` message, σ collects in a set U_σ all the ids it does not know (i_j is in U_σ iff σ does not know i_j), and sends U_σ back to β by means of a `BatchAcquired` message (fig. 1, step 5). Upon receiving σ's `BatchAcquired` message, β builds a set A_σ containing all id assignments that σ is missing (a_j is in A_σ iff i_j is in U_σ). Having done so, β sends a `Signatures` message to σ (fig. 1, step 6). The `Signatures` message contains the batch's multi-signature m, the straggler set S, and A_σ. We underline the importance of sending id assignments upon request only. Thinking to shave one round-trip off the protocol, β could naively package in a single message all submissions, all (multi-)signatures, and all assignments relevant to the batch. In doing so, however, β would force each server to receive

[10] For example, using BLS, β could aggregate the public keys of $\chi_1, \ldots, \chi_{n-1}, \chi_{n+1}, \ldots, \chi_b$ into a public key $\tilde{t}_n$, then show that the aggregation of $\tilde{t}_n$ with χ's public key correctly verifies m against r. Doing so, however, would additionally require β to exhibit a proof that $\tilde{t}_n$ is not a rogue public key, i.e., that $\tilde{t}_n$ indeed results from the aggregation of client public keys. This could be achieved by additionally having $\chi_1, \ldots, \chi_b$ multi-sign some hard-coded statement to prove that they are not rogues. β could aggregate such signatures on the fly, producing a rogue-resistance proof for $\tilde{t}_n$ that can be transmitted and verified in constant time. This, however, is expensive (and, frankly, at the limit of our cryptographic expertise).

one assignment per submission, immediately forfeiting Draft's oracular efficiency. At the batching limit we assume that all servers already know all broadcasting clients. In that case, both U_σ and A_σ are constant-sized, empty sets, adding only a vanishing amount of communication complexity to the protocol. Upon receiving the `Signatures` message, σ verifies and learns all assignments in A_σ. Having done so, σ knows $i_1, \ldots, i_b$. As we outlined above, σ can now efficiently authenticate the whole batch, verifying m against the batch's root r, and each i_j in S against p_j. Having established the integrity of the whole batch, σ produces a *witness shard* for the batch, i.e., a multi-signature w_σ for $[\texttt{Witness}, r]$, effectively affirming to have successfully authenticated the batch. σ sends w_σ back to β by means of a `WitnessShard` message (fig. 1, step 7). Having received a valid `WitnessShard` message from $f+1$ servers, β aggregates all witness shards into a *witness* w. Because w is a plurality ($f+1$) certificate, at least one correct server necessarily produced a witness shard for the batch. This means that at least one correct server has successfully authenticated the batch by means of client (multi-)signatures. Because w could not have been gathered if the batch was not properly authenticated, w itself is sufficient to authenticate the batch, and β can drop all (now redundant) client-generated (multi-)signatures for the batch. Unlike m, w is easy to verify, as it is signed by only $f+1$, globally known servers. Like m, w authenticates r. As such, any p_j can now be authenticated just by producing w, and (i_j, p_j)'s proof of inclusion q_j.

Gathering a commit certificate. Having successfully gathered a witness w for the batch, β sends w to all servers by means of a `Witness` message (fig. 1, step 8). Upon receiving the `Witness` message, σ moves on to check $(i_1, p_1), \ldots, (i_b, p_b)$ for equivocations. More precisely, σ builds a set of *exceptions* E_σ containing the ids of all equivocating submissions in the batch (i_j is in E_σ iff σ previously observed χ_j submit a payload p'_j that conflicts with p_j; we recall that p_j and p'_j conflict if their contexts are the same, but their messages are different). σ then produces a *commit shard* for the batch, i.e., a multi-signature c_σ for $[\texttt{Commit}, r, E_\sigma]$, effectively affirming that σ has found all submissions in the batch to be non-equivocated, *except* for those in E_σ. In the good case, every client is correct and E_σ is empty. Having produced c_σ, σ moves on to build a set Q_σ containing a *proof of equivocation* for every element in E_σ. Let us assume that σ previously received from some χ_k a payload p'_k that conflicts with p_k. σ must have received p'_k as part of some witnessed batch. Let r'_k identify the root of p'_k's batch, let w'_k identify r'_k's witness, let q'_k be (i_k, p'_k)'s proof of inclusion in r'_k. By exhibiting (r'_k, w'_k, p'_k), σ can prove to β that χ_k equivocated: p_k conflicts with p'_k, and (i_k, p'_k) is provably part of a batch whose integrity was witnessed by at least one correct server. Furthermore, because correct clients never equivocate, (r'_k, w'_k, p'_k) is sufficient to convince β that χ_k is Byzantine. For each i_j in E_σ, σ collects in Q_σ a proof of equivocation $\left(r'_j, w'_j, p'_j\right)$. Finally, σ sends a `CommitShard` message back to β (fig. 1, step 9). The `CommitShard` message contains c_σ, E_σ and Q_σ. Upon receiving σ's `CommitShard` message, σ verifies c_σ against r and E_σ, then checks all proofs in Q_σ. Having collected valid `CommitShard` messages from a quorum of servers $\sigma_1, \ldots, \sigma_{2f+1}$, β aggregates all commit shards into a *commit certificate* c. We underline that each σ_j signed the same root r, but a potentially different set of exceptions E_{σ_j}. Let E denote the union of $E_{\sigma_1}, \ldots, E_{\sigma_{2f+1}}$. We call E the batch's *exclusion* set. Because a proof of equivocation cannot be produced against a correct client, β knows that all clients identified by E are necessarily Byzantine. In particular, because χ is correct, $(i = i_n)$ is guaranteed to not be in E.

Committing the batch. Having collected a commit certificate c for the batch, β sends c to all servers by means of a `Commit` message (fig. 1, step 10). Upon receiving the `Commit` message, σ verifies c, computes the exclusion set E, then delivers every payload p_j whose id i_j is not in E. Recalling that c is assembled from a quorum of commit shards, at least $f+1$ correct servers contributed to c. This means that, if some i_k is not in E, then at least $f+1$ correct servers found p_k not to be equivocated. As in most BRB implementations [9], this guarantees that no two commit certificates can be gathered for equivocating payloads: Draft's consistency is upheld.

The role of equivocation proofs. As the reader might have noticed, β does not attach any proof of equivocation to its `Commit` message. Having received β's commit certificate c, σ trusts β's exclusion set E, ignoring every payload whose id is in E. This is not because σ can trust β to uphold validity. On the contrary, σ has *no way* to determine that β is not maliciously excluding the payload of a correct client. Indeed, even if σ were to verify a proof of exclusion for every element in E, a malicious β could still censor a correct client simply by ignoring its `Submit` message in the first place. Equivocation proofs are fundamental to Draft's validity not because they *force* malicious brokers to uphold validity, but because they *enable* correct brokers to do the same. Thanks to equivocation proofs, a malicious server cannot trick a correct broker into excluding the payload of a correct client. This is enough to guarantee validity. As we discuss below, χ successively submits p to all brokers until it receives a certificate attesting that p was delivered by at least one correct server. Because we assume at least one broker to be correct, χ is eventually guaranteed to succeed.

Notifying the clients. Having delivered every payload whose id is not in the exclusion set E, σ produces a *completion shard* for the batch, i.e., a multi-signature z_σ for $[\texttt{Completion}, r, E]$, effectively affirming that σ has delivered all submissions in the batch whose id is not in E. σ sends z_σ to β by means of a `CompletionShard` message (fig. 1, step 11). Upon receiving $f+1$ valid `CompletionShard` messages, β assembles all completion shards into a *completion certificate* z. Finally, β sends a `Completion` message to $\chi_1, \ldots, \chi_b$ (fig. 1, step 12). The `Completion` message contains z and E. Upon receiving the `Completion` message, χ verifies z against E, then checks that i is not in E. Because at least one correct server contributed a completion shard to z, at least one correct process delivered all payloads that E did not exclude, including p. Having succeeded in broadcasting p, χ does not need to engage further, and can stop successively submitting p to all brokers.

No one is left behind. As we discussed above, upon receiving the commit certificate c, σ delivers every payload in the batch whose id is not in the exclusion set E. Having gotten at least one correct server to deliver the batch, β is free to disengage, and moves on to assembling and brokering its next batch. In a moment of asynchrony, however, all communications between β and $\tilde{\sigma}$ might be arbitrarily delayed. This means that $\tilde{\sigma}$ has no way of telling whether or not it will eventually receive batch and commit certificate: a malicious β might have deliberately left $\tilde{\sigma}$ out of the protocol. Server-to-server communication is thus required to guarantee totality. Having delivered the batch, σ waits for an interval of time long enough for all correct servers to deliver batch and commit certificate, *should the network be synchronous and β correct.* σ then sends to all servers an `OfferTotality` message (fig. 1, step 13). The `OfferTotality` message contains the batch's root r, and the exclusion set E. In the good case, upon receiving σ's `OfferTotality` message, every server has delivered the batch and ignores the offer. This, however, is not the case for slow $\hat{\sigma}$, which replies to

σ with an `AcceptTotality` message (fig. 1, step 14). Upon receiving $\tilde{\sigma}$'s `AcceptTotality` message, σ sends back to $\tilde{\sigma}$ a `Totality` message (fig. 1, step 15). The `Totality` message contains all submissions $(i_1, p_1), \ldots, (i_b, p_b)$, id assignments for $i_1, \ldots, i_b$, and the commit certificate c. Upon delivering σ's `Totality` message, $\tilde{\sigma}$ computes r from $(i_1, p_1), \ldots, (i_b, p_b)$, checks c against r, computes E from c, and delivers every payload p_j whose id i_j is not in E. This guarantees totality and concludes the protocol.

4.2 Complexity

Directory density. As we introduced in Section 4.1, Draft uses ids assigned by its underlying Directory (DIR) abstraction to identify payload senders. A DIR-assigned id is composed of two parts: a *domain* and an *index*. Domains form a finite set $\mathbb{D}$ whose size does not increase with the number of clients, indices are natural numbers. Along with safety (e.g., no two processes have the same id) and liveness (e.g., every correct client that requests an id eventually obtains an id), DIR guarantees *density*: the index part of any id is always smaller than the total number of clients c (i.e., each id index is between 0 and $(c-1)$). Intuitively, this echoes the (stronger) density guarantee provided by Oracle-CSB, the oracle-based implementation of CSB we introduced in Section 1 to bound Draft's performance. In Oracle-CSB, the oracle organizes all clients in a list, effectively labeling each client with an integer between 0 and $(c-1)$. In a setting where consensus cannot be achieved, agreeing on a totally-ordered list of clients is famously impossible: a consensus-less DIR implementation cannot assign ids if $|\mathbb{D}| = 1$. However, DIR can be implemented without consensus if servers are used as domains ($\mathbb{D} = \Sigma$). In our DIR implementation Dibs, each server maintains an independent list of public keys. In order to obtain an id, a client χ has each server add its public key to its list, then selects a server σ to be its *assigner*. In doing so, χ obtains an id (σ, n), where $n \in 0..(c-1)$ is χ's position in σ's log. In summary, a consensus-less implementation of DIR still guarantees that indices will be smaller than c, at the cost of a non-trivial domain component for each id. This inflates the size of each individual id by $\lceil \log_2(|\mathbb{D}|) \rceil$ bits.

Batching ids. While DIR-assigned ids come with a non-trivial domain component, the size overhead due to domains vanishes when infinitely many ids are organized into a batch. This is because domains are constant in the number of clients. Intuitively, as infinitely many ids are batched together, repeated domains become compressible. When building a batch, a Draft broker represents the set I of sender ids not as a list, but as a map $\tilde{i}$. To each domain, $\tilde{i}$ associates all ids in I under that domain (n is in $\tilde{i}[d]$ iff (d, n) is in I). Because $\tilde{i}$'s keys are fixed, as the size of I goes to infinity, the bits required to represent $\tilde{i}$'s keys are completely amortized by those required to represent $\tilde{i}$'s values. At the batching limit, the cost of representing each id in $\tilde{i}$ converges to that of representing its index only, $\lceil \log_2(c) \rceil$.

Protocol cost. At the batching limit we assume a good-case execution: links are synchronous, all processes are correct, and the set of brokers contains only one element. We additionally assume that infinitely many clients broadcast concurrently. Finally, we assume all servers to already know all broadcasting clients. Let β denote the only broker. As all broadcasting clients submit their payloads to β within a suitably narrow time window, β organizes all submissions into a single batch with root r. Because links are synchronous and all clients are correct, every broadcasting client submits its multi-signature for r in time. Having removed all individual signatures from the batch, β is left with a single, aggregated multi-signature m and an empty straggler set S. β compresses the sender ids and disseminates the batch to

all servers. As m and S are constant-sized, the amortized cost for a server to receive each payload p is $(\lceil \log_2(c) \rceil + |p|)$ bits. As m authenticates the entire batch, a server authenticates each payload at an amortized cost of 0 signature verifications. The remainder of the protocol unfolds as a sequence of constant-sized messages: because all broadcasting clients are known to all servers, no server requests any id assignment; witnesses are always constant-sized; and because all processes are correct, no client equivocates and all exception sets are empty. Finally, again by the synchrony of links, all offers of totality are ignored. In summary, at the batching limit a server delivers a payload at an amortized cost of 0 signature verifications and $(\lceil \log_2(c) \rceil + |p|)$ bits exchanged.

Latency. As depicted in Figure 1, the latency of Draft is 10 message delays in the synchronous case (fast servers deliver upon receiving the broker's Commit message), and at most 13 message delays in the asynchronous case (slow servers deliver upon receiving other servers' Totality messages). By comparison, the latency of the optimistic reliable broadcast algorithm by Cachin *et al.* [13] is respectively 4 message delays (synchronous case) and 6 message delays (asynchronous case). Effectively, Draft trades oracular efficiency for a constant latency overhead.

Worst-case complexity. In the worst case, a Draft server delivers a b-bits payload by exchanging $O((\log(c) + b)kn)$ bits, where c, k and n respectively denote the number of clients, brokers and servers. In brief, the same id, payload and signature is included by each broker in a different batch (hence the k term) and propagated in an all-to-all fashion (carried by `Totality` messages) across correct servers (hence the n term). By comparison, the worst-case communication complexity of Cachin *et al.*'s optimistic reliable broadcast is $O(ln)$ per server, where l is the length of the broadcast payload. A direct batched generalization of the same algorithm, however, would raise the worst-case communication to $O(ln^2)$ per server, similar to that of Draft when $n \sim k$. Both batched Bracha and Draft can be optimized by polynomial encoding, reducing their per-server worst-case complexity to $O(ln)$ and $O((\log(c) + b)k)$ respectively. Doing so for Draft, however, is beyond the scope of this paper.

5 Conclusions

Our contributions. In this paper we study Client-Server Byzantine Reliable Broadcast (CSB), a multi-shot variant of Byzantine Reliable Broadcast (BRB) whose interface is split between broadcasting clients and delivering servers. We introduce Oracle-CSB, a toy implementation of CSB that relies on a single, infallible oracle to uphold all CSB properties. Unless clients can be assumed to broadcast at a non-uniform rate, Oracle-CSB's signature and communication complexities are optimal: in Oracle-CSB, a server delivers a payload p by performing 0 signature verifications, and exchanging $(\lceil \log_2(c) \rceil + |p|)$ bits, where c is the number of clients. We present Draft, our implementation of CSB. Draft upholds all CSB properties under classical BRB assumptions (notably asynchronous links and less than a third of faulty servers). When links are synchronous and all processes are correct, however, and at the limit of infinite concurrently broadcasting clients, Draft's signature and communication complexities match those of Oracle CSB.

Future work. We hope to extend Draft to allow multiple messages by the same client in the same batch. We envision that this could be achieved by using other types of cryptographic accumulators or variants of Merkle trees, such as Merkle-Patricia trees [25]. It would also be interesting to see if the worst-case performance of Draft could be improved, e.g. by using error correction codes (ECC) or erasure codes, without significantly affecting its good-case performance. Lastly, we hope to use Draft's keys ideas to implement a total-order broadcast primitive, improving the scalability of existing SMR implementations.

References

1. Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery.
2. Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct erasure coding proof systems. *Cryptology ePrint Archive*, 2021.
3. 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 '18, New York, NY, USA, 2018. Association for Computing Machinery.
4. Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Byzantine-tolerant causal broadcast. *Theoretical Computer Science*, 885:55–68, 2021.
5. Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. Efficient algorithms for pairing-based cryptosystems. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, pages 354–369, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
6. Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.
7. Joseph T.A. Birman K.P. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.
8. Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.
9. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
10. Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *JACM*, 32(4), 1985.
11. Christian Cachin. State machine replication with byzantine faults. In *Replication*, pages 169–184. Springer, 2010.
12. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
13. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
14. Christian Cachin and Jonathan A. Poritz. Secure Intrusion-tolerant Replication on the Internet. In *DSN*, 2002.
15. Christian Cachin and Jonathan A Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings International Conference on Dependable Systems and Networks*, pages 167–176. IEEE, 2002.

16 Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *Proceedings of the 24th Symposium on Reliable Distributed Systems – SRDS 2005*, pages 191–202, October 2005.

17 Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira. Oracular Byzantine Reliable Broadcast (Extended Version), 2022.

18 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

19 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2020. doi:10.1109/DSN48063.2020.00023.

20 Thomas Cover and Joy Thomas. *Elements of Information Theory, Second Edition.* John Wiley & Sons, 2005.

21 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly Blockchain. *CoRR*, 2018.

22 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)*, pages 466–483. IEEE, 2021.

23 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*, pages 34–50, 2022.

24 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

25 Haitz Sáez de Ocáriz Borde. An overview of trees in blockchain technology: Merkle trees and merkle patricia tries, 2022.

26 Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

27 Danny Dolev and Rüdiger Reischuk. Bounds on Information Exchange for Byzantine Agreement. *J. ACM*, 32(1):191–204, January 1985.

28 Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.

29 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos Seredinschi. The Consensus Number of a Cryptocurrency. In *PODC*, 2019.

30 James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, 2007.

31 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*, pages 165–175, 2021.

32 Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming*, pages 204–215. Springer, 2005.

33 Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

34 D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a wan. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 87–94, 1997.

35 Dahlia Malkhi and Michael Reiter. A High-Throughput Secure Reliable Multicast Protocol. *Journal of Computer Security*, 5:113–127, 1996.

36 Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.

37 Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

38 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

39 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

40 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

41 James S Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*, pages 173–180. IEEE, 2006.

42 HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

43 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

44 Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3), 1994.

45 Nuno Santos and André Schiper. Optimizing Paxos with batching and pipelining. *Theoretical Computer Science*, 496:170–183, 2013. Distributed Computing and Networking (ICDCN 2012).

46 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

47 Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 17–33, 2022.

48 Andrew Chi-Chih Yao. Some Complexity Questions Related to Distributive Computing. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 209–213, New York, NY, USA, 1979. Association for Computing Machinery.